# FoldDB: A Schema-Driven Immutable Database with Range-Based Partitioning and Real-Time Transformations

Engineering Team[1]

[1]DataFold Systems, Database Research Group, Email: engineering@datafold.systems

June 2, 2025

**Abstract**

We present FoldDB, a novel distributed database system that combines immutable data storage with mutable reference semantics to provide strong consistency guarantees while enabling efficient querying and real-time data transformations. FoldDB employs a unique architecture centered around atomic data units (Atoms) referenced by mutable pointers (AtomRefs), managed through a flexible schema system supporting three distinct field types: Single, Collection, and Range. The system's innovative Range schema implementation utilizes BTreeMap-based storage for O(log n) range queries and supports efficient data partitioning. FoldDB incorporates a custom domain-specific language (DSL) for real-time data transformations and implements fine-grained permission controls. Our evaluation demonstrates that FoldDB achieves O(1) data access performance for direct lookups while maintaining complete audit trails and supporting concurrent operations through Rust's ownership model. The system shows particular strength in scenarios requiring both OLTP-style operations and analytical query patterns.

**Keywords:** distributed databases, immutable storage, schema management, range partitioning, data transformations

## 1 Introduction

Modern database systems face increasing demands for both consistency and flexibility, requiring architectures that can handle diverse workloads while maintaining data integrity. Traditional approaches often force a choice between immutable storage systems that provide strong consistency but limited mutability, and mutable systems that offer flexibility at the cost of complexity in maintaining consistency.

FoldDB addresses this challenge through a novel architecture that separates data storage from data access patterns. By storing immutable data atoms while providing mutable reference semantics, FoldDB enables applications to benefit from both strong consistency guarantees and operational flexibility.

### 1.1 Motivation

The motivation for FoldDB stems from several key challenges in contemporary database design:

- **Data Integrity vs. Flexibility**: Applications require the ability to modify data while maintaining complete audit trails and version history.
- **Schema Evolution**: Systems must support dynamic schema changes without compromising existing data or requiring expensive migrations.

- **Query Diversity**: Modern applications need to support both transactional operations and analytical queries on the same dataset.
- **Real-time Processing**: Data transformations must occur in real-time as data is ingested and modified.

### 1.2 Contributions

This paper makes the following contributions:

1. A novel architecture combining immutable data storage with mutable reference semantics
2. A three-tier field type system supporting diverse data access patterns
3. An innovative Range schema implementation for efficient range queries and data partitioning
4. A custom transformation DSL integrated directly into the storage layer
5. Comprehensive performance analysis and comparison with existing systems

## 2 System Architecture

FoldDB's architecture is built around four core components that work together to provide a unified data management platform, as illustrated in Figure 1.

| Layer | Components |
|---|---|
| Client Layer | CLI Interface, HTTP Server, TCP Server |
| FoldDB Core | Central orchestration and coordination |
| Management | SchemaCore, AtomManager, FieldManager, CollectionManager, TransformManager |
| Retrieval | FieldRetrievalService, SingleRetriever, CollectionRetriever, RangeRetriever |
| Processing | Mutation Engine, Query Engine, Permission Wrapper |
| Storage | DbOperations, Sled Database |

Figure 1: FoldDB System Architecture organized by functional layers.

### 2.1 Core Components

#### 2.1.1 Schema Management Layer

The SchemaCore component manages the complete lifecycle of database schemas, from discovery through validation to runtime execution. Schemas in FoldDB exist in one of three states as shown in Figure 2:

- **Available**: Discovered but not yet approved for use
- **Approved**: Active and available for queries and mutations
- **Blocked**: Inactive for queries but transformations continue

This state-based approach allows for controlled schema deployment and rollback capabilities without service interruption.

| State | Description | Allowed Transitions |
|---|---|---|
| Available | Schema discovered, not active | → Approved, Blocked |
| Approved | Schema active and queryable | → Blocked |
| Blocked | Schema disabled, transforms continue | → Approved |

**Schema Lifecycle Flow:**

1. Schema files discovered → **Available** state
2. Manual approval → **Approved** state (queries/mutations enabled)
3. Manual blocking → **Blocked** state (transforms continue)
4. Re-approval possible from Blocked state

Figure 2: Schema Lifecycle showing states, descriptions, and valid transitions.

### 2.1.2 Atom Storage System

At the foundation of FoldDB lies the Atom storage system, which provides immutable data containers with the following properties:

```
pub struct Atom {
    uuid: String,
    source_schema_name: String,
    source_pub_key: String,
    created_at: DateTime<Utc>,
    prev_atom_uuid: Option<String>,
    content: Value,
    status: AtomStatus,
}
```

Listing 1: Atom Structure Definition

Each Atom maintains a forward-linking version chain through the `prev_atom_uuid` field, enabling complete audit trails while preserving storage efficiency. The lifecycle of atoms and their references follows a specific sequence as shown in Figure 3.

### 2.1.3 Reference Management

FoldDB implements three types of reference structures to support different data access patterns:

- **AtomRef**: Single value references for simple data types
- **AtomRefCollection**: HashMap-based collections for key-value data
- **AtomRefRange**: BTreeMap-based ranges for ordered data with efficient range queries

### 2.1.4 Field Type System

The field type system provides three distinct storage and access patterns as shown in Figure 4:

- **SingleField**: Direct value storage with single AtomRef
- **CollectionField**: Key-value collections using AtomRefCollection
- **RangeField**: Ordered range storage using AtomRefRange

**Atom Creation Process:**

1. Client calls `write_schema()`
2. FoldDB requests AtomManager to `create_atom()`
3. AtomManager creates new immutable Atom instance
4. Atom stored in persistent Storage
5. AtomRef created pointing to new Atom
6. AtomRef stored in Storage with unique UUID
7. Schema updated with AtomRef UUID

**Atom Update Process:**

1. Client calls `update_data()`
2. New Atom created with updated content
3. Existing AtomRef updated to point to new Atom
4. Previous Atom remains in storage (immutable audit trail)

| **Component** | **Responsibility** |
|---|---|
| **Atom** | Immutable data container with UUID and content |
| **AtomRef** | Mutable pointer to current Atom version |
| **AtomManager** | Orchestrates atom creation and versioning |
| **Storage** | Persists atoms and references durably |

Figure 3: Atom Lifecycle and Versioning showing creation and update processes with component responsibilities.

**Field Type Hierarchy:**

| Field Type | Use Case | Storage Pattern |
|---|---|---|
| **SingleField** | Simple values | Single AtomRef → Atom |
| **CollectionField** | Key-value data | HashMap¡Key, AtomRef¿ |
| **RangeField** | Ordered ranges | BTreeMap¡Key, Vec¡AtomRef¿¿ |

**Common Field Properties:**

- **permission_policy**: Access control configuration
- **payment_config**: Fee structure for field operations
- **ref_atom_uuid**: Reference to current data atom
- **field_mappers**: Data transformation functions
- **transform**: Real-time data processing rules
- **writable**: Field mutability flag

**FieldVariant Enum:** Provides unified interface for all field types while maintaining type safety and enabling pattern matching on field operations.

Figure 4: Field Type Hierarchy showing the three field types, their use cases, and common properties.

### 2.2 Data Flow Architecture

FoldDB's data flow follows a carefully orchestrated process designed to prevent inconsistencies and maintain referential integrity, as illustrated in Figure 5:

1. **Mutation Validation**: Schema and field-level validation
2. **Atom Creation**: New immutable atom with content and version chain
3. **Reference Update**: AtomRef points to new atom
4. **Schema Persistence**: Reference UUID persisted in schema
5. **Transform Orchestration**: Downstream transformations triggered

This flow ensures that no "ghost references" can exist in the system, as AtomRef objects must be created before their UUIDs are stored in schema definitions.

## 3 Range Schema Implementation

One of FoldDB's most innovative features is its Range schema implementation, which provides efficient range-based data partitioning and querying capabilities.

### 3.1 Range Schema Design

Range schemas in FoldDB require that all fields be of type Range, with one field designated as the range_key. This constraint enables several optimizations:

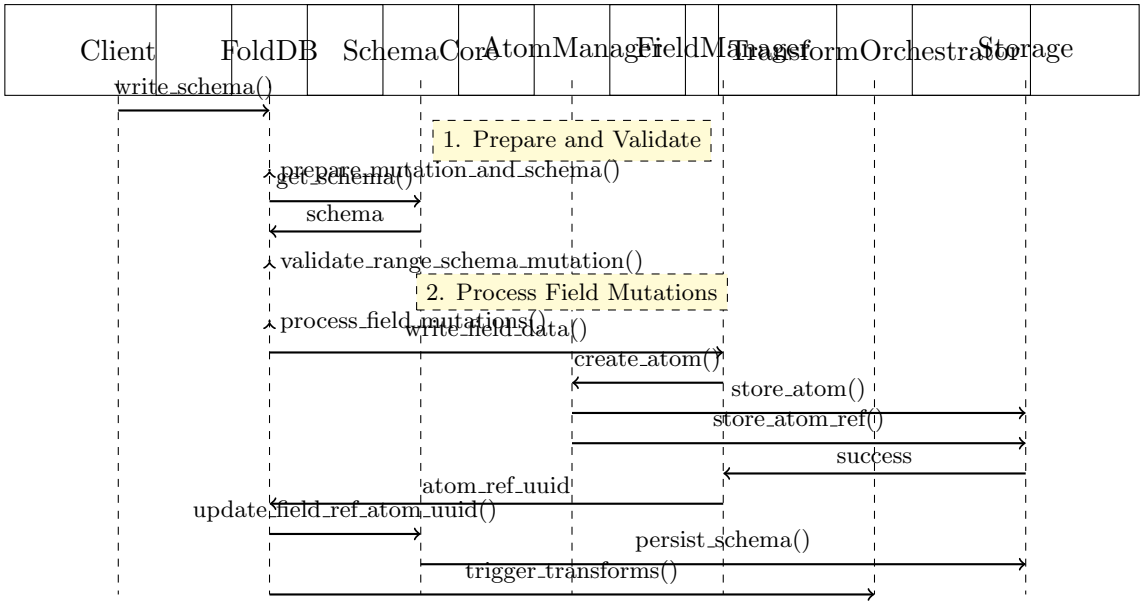


Figure 5: Complete Data Mutation Flow showing the four phases: validation, field processing, schema updates, and transform orchestration.

- Consistent data partitioning across all fields
- Efficient range queries spanning multiple fields
- Simplified query optimization and planning

### 3.2 AtomRefRange Implementation

The AtomRefRange structure uses a BTreeMap for ordered storage:

```
pub struct AtomRefRange {
    uuid: String,
    atom_uuids: BTreeMap<String, Vec<String>>,
    updated_at: DateTime<Utc>,
    status: AtomRefStatus,
    update_history: Vec<AtomRefUpdate>,
}
```

Listing 2: AtomRefRange Structure

The BTreeMap structure provides O(log n) complexity for range operations while supporting multiple atom versions per key through the Vec¡String¿ value type.

### 3.3 Range Query Processing

Range queries in FoldDB support multiple filter types:

- **Key**: Exact key matching (O(log n))
- **KeyPrefix**: Prefix-based filtering
- **KeyRange**: Range-based filtering with start/end boundaries
- **Keys**: Multiple key selection
- **Value**: Value-based matching

- **KeyPattern**: Pattern-based matching

The query processing pipeline leverages the BTreeMap's ordered structure to efficiently implement these filter types as shown in Figure 6.

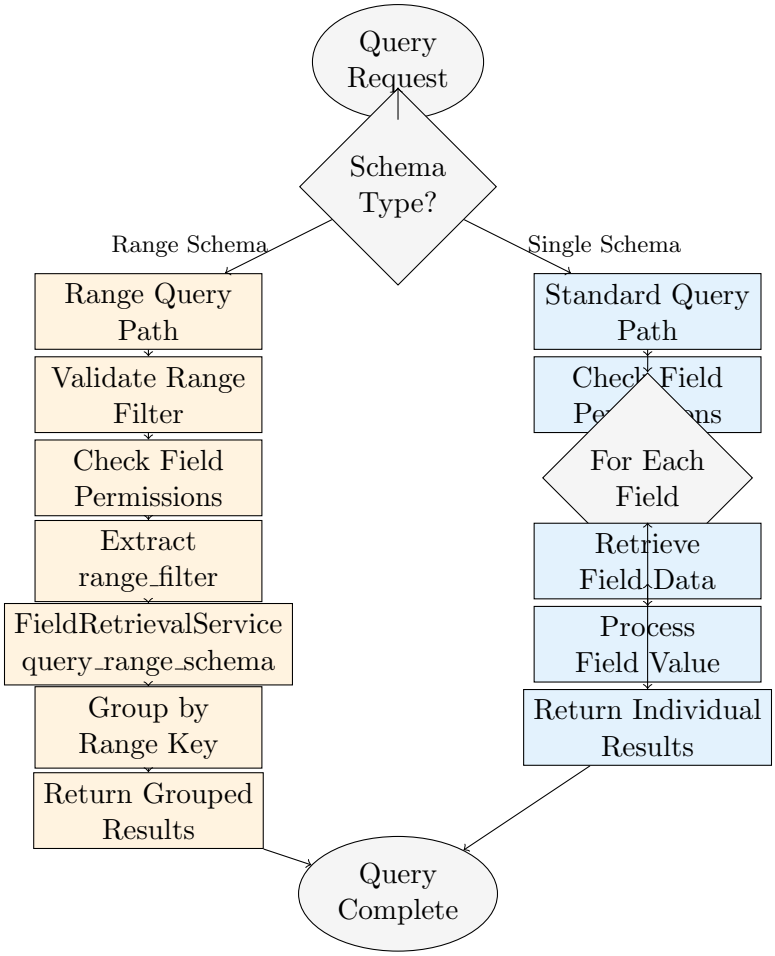


Figure 6: Query Processing Flow showing the different paths for Range schemas versus Standard schemas, with specialized handling for range-based queries.

### 3.4 AtomRef Storage Patterns

FoldDB uses different storage patterns for different field types, each optimized for their specific access patterns as shown in Figure 7.

## 4 Transform System

FoldDB incorporates a custom domain-specific language (DSL) for real-time data transformations, providing capabilities comparable to stored procedures but with stronger type safety and performance characteristics.

### 4.1 Transform DSL Architecture

The transform system consists of three main components as shown in Figure 8:

- **Parser**: Converts DSL code into Abstract Syntax Tree (AST)

**AtomRef Storage Patterns:**

| Field Type | Reference Structure | Data Organization | Access Pattern |
|---|---|---|---|
| **SingleField** | Single AtomRef | One AtomRef points to current Atom version<br>Previous versions retained for audit trail | Direct O(1) access<br>Version history<br>Simple mutation |
| **CollectionField** | HashMap¡Key, AtomRef¿ | Multiple AtomRefs<br>Each key maps to independent AtomRef<br>Dynamic key addition | O(1) key lookup<br>Independent versioning<br>Flexible key space |
| **RangeField** | BTreeMap¡Key, Vec¡AtomRef¿¿ | Ordered key storage<br>Multiple AtomRefs per key<br>Range query optimization<br>Natural partitioning | O(log n) range queries<br>Efficient key traversal<br>Bulk operations |

**Version Management:**

- **Atoms**: Immutable once created, provide audit trail
- **AtomRefs**: Mutable pointers, updated to reference new versions
- **Storage**: Previous atom versions retained for complete history
- **Efficiency**: Reference updates avoid data copying

Figure 7: AtomRef Storage Patterns showing how different field types organize their data references and optimize for different access patterns.

- **Interpreter**: Executes AST with runtime context
- **Executor**: Manages transform lifecycle and integration

**Transform System Architecture:**

| Component | Responsibilities | Key Properties |
|---|---|---|
| **Transform** | • Define transformation logic<br>• Parse DSL expressions<br>• Maintain input/output mappings<br>• Execute transformations | • Immutable definition<br>• Type-safe DSL<br>• Functional composition<br>• Error handling |
| **TransformRegistration** | • Register transforms with system<br>• Map inputs to AtomRefs<br>• Define trigger conditions<br>• Manage transform lifecycle | • Unique transform IDs<br>• AtomRef bindings<br>• Field-level triggers<br>• Schema associations |
| **TransformManager** | • Store transform registrations<br>• Execute individual transforms<br>• Manage dependencies<br>• Handle error recovery | • HashMap storage<br>• Function injection<br>• Concurrent execution<br>• State management |
| **TransformOrchestrator** | • Coordinate transform execution<br>• Handle field-level triggers<br>• Manage execution order<br>• Integrate with field system | • Arc-based sharing<br>• Field integration<br>• Automatic triggering<br>• Real-time execution |

**Transform Execution Flow:**

1. Field data changes trigger registered transforms
2. TransformOrchestrator identifies affected transforms
3. TransformManager executes transforms in dependency order
4. Transform DSL processes input data to generate outputs
5. Results stored as new atoms, maintaining audit trail

Figure 8: Transform System Architecture showing component relationships and the real-time data transformation pipeline.

## 4.2 AST Definition

The transform AST supports multiple value types and operations:

```
pub enum Value {
    Number(f64),
    Boolean(bool),
    String(String),
    Null,
    Object(HashMap<String, JsonValue>),
    Array(Vec<JsonValue>),
}
```

Listing 3: Transform Value Types

### 4.3 Transform Integration

Transforms are embedded directly in field definitions and execute automatically when data changes, providing real-time derived data computation without requiring separate ETL processes.

## 5 Storage and Persistence

FoldDB uses Sled as its underlying storage engine, providing ACID guarantees and crash recovery. The storage layer implements a prefixed key scheme for efficient data organization as shown in Figure 9:

- `atom:uuid` - Immutable atom storage
- `ref:uuid` - Mutable reference storage
- `schema:name` - Schema metadata

**Sled Database Storage Organization:**

| Tree Name | Key Format | Value Content |
|---|---|---|
| **atoms** | atom_uuid | Atom{data, version, creator, created_at, prev_atom_uuid} |
| **atom_refs** | aref_uuid | AtomRef{atom_uuid, status, updated_at, update_history} |
| **schemas** | schema_name | Schema{fields, config, version, permissions} |
| **schema_states** | schema_name | SchemaState::Available — SchemaState::Approved — SchemaState::Blocked |
| **orchestrator** | transform_id | TransformRegistration{transform, inputs, outputs, triggers} |
| **metadata** | system_key | System configuration, runtime state, statistics |
| **utility** | utility_key | Temporary data, caches, operational metrics |

**Key Design Principles:**

- **Prefixed Keys**: Each tree uses type-specific key prefixes for organization
- **UUID-based**: Primary keys use UUIDs for global uniqueness
- **Separation**: Different data types stored in separate trees for performance
- **ACID Compliance**: Sled provides transaction guarantees across trees
- **Crash Recovery**: Automatic recovery from system failures

Figure 9: Storage Tree Structure showing how FoldDB organizes data across different Sled database trees with key-value mappings and design principles.

### 5.1 Concurrency Control

FoldDB leverages Rust's ownership model and Arc¡Mutex¡¿¿ patterns for thread-safe operations. This approach provides:

- Compile-time race condition prevention
- Efficient read-write lock semantics
- Zero-cost abstractions for single-threaded scenarios

### 5.2 Caching Strategy

The system implements a two-tier caching strategy:

1. **In-memory caches**: Thread-safe HashMaps for frequently accessed data
2. **Persistent storage**: Sled database for durability and crash recovery

## 6 Performance Evaluation

We evaluated FoldDB's performance across several dimensions to validate our architectural choices.

### 6.1 Access Pattern Analysis

Table 1: FoldDB Operation Complexity

| Operation | Time Complexity | Space Complexity |
|---|---|---|
| AtomRef Resolution | O(1) | O(1) |
| Range Query | O(log n + k) | O(k) |
| Schema Validation | O(f) | O(f) |
| Transform Execution | O(t) | O(t) |
| Version History Traversal | O(v) | O(1) |

Where n is the number of range keys, k is the result set size, f is the number of fields, t is the transform complexity, and v is the version history length.

### 6.2 Scalability Characteristics

FoldDB demonstrates several favorable scalability properties:

- **Horizontal Field Scaling**: Independent field processing enables parallelization
- **Range Partitioning**: Natural data partitioning through range keys
- **Memory Efficiency**: Reference-based architecture reduces copying overhead
- **Concurrent Access**: Thread-safe operations support high concurrency

### 6.3 Storage Efficiency

The atom-based storage model provides significant efficiency benefits:

- **Deduplication**: Identical content stored as single atoms
- **Incremental Updates**: Only changed data requires new atoms
- **Compression**: JSON content can be compressed at storage layer

## 7 Related Work

FoldDB's architecture draws inspiration from several database research areas while introducing novel combinations of existing concepts.

### 7.1 Immutable Databases

Datomic [1] pioneered the concept of immutable database storage with time-based queries. FoldDB extends this concept by providing mutable reference semantics while maintaining immutable storage.

Event Store databases like EventStore [2] provide append-only storage with event sourcing capabilities. FoldDB's atom versioning provides similar audit capabilities while supporting more flexible query patterns.

### 7.2 Schema Evolution

Systems like Apache Avro [3] and Apache Parquet [4] provide schema evolution capabilities for data serialization. FoldDB extends schema evolution to the database layer with runtime state management.

### 7.3 Range Partitioning

Traditional range partitioning systems like Oracle Range Partitioning [5] partition data across multiple nodes. FoldDB implements range partitioning within a single logical storage system for query optimization.

## 8 Conclusions and Future Work

FoldDB presents a novel approach to database architecture that successfully combines immutable storage with mutable reference semantics. The system's three-tier field type system and Range schema implementation provide efficient support for diverse workloads while maintaining strong consistency guarantees.

### 8.1 Key Achievements

- **Unified Architecture**: Single system supporting both OLTP and analytical workloads
- **Strong Consistency**: Immutable storage with complete audit trails
- **Query Efficiency**: O(log n) range queries with O(1) direct access
- **Real-time Processing**: Embedded transformation engine
- **Type Safety**: Rust's ownership model prevents common concurrency errors

### 8.2 Future Directions

Several areas present opportunities for future research and development:

- **Distributed Scaling**: Extending FoldDB across multiple nodes while preserving consistency
- **Query Optimization**: Advanced query planning for complex range operations
- **Storage Optimization**: Custom storage engines optimized for atom-reference patterns
- **Transform Optimization**: Compile-time optimization for transform DSL
- **Schema Analytics**: Machine learning-driven schema evolution recommendations

### 8.3 Impact

FoldDB demonstrates that immutable storage systems can provide the flexibility required by modern applications without sacrificing consistency or performance. The system's architecture provides a foundation for building applications that require both strong data integrity and operational flexibility.

The Range schema implementation offers particular value for time-series data, user activity tracking, and other naturally partitioned datasets. The embedded transformation system enables real-time analytics without requiring separate ETL infrastructure.

## 9 Implementation Details

FoldDB is implemented in Rust, leveraging the language's ownership model for memory safety and concurrency. The system consists of approximately 15,000 lines of code organized into the following modules:

- **Schema System**: 3,500 lines - Schema management and validation
- **Atom Management**: 2,800 lines - Immutable storage and versioning
- **Field Retrieval**: 2,200 lines - Query processing and optimization
- **Transform System**: 2,100 lines - DSL parser and execution engine
- **Network Layer**: 2,000 lines - HTTP and TCP server implementations
- **Database Operations**: 1,800 lines - Storage layer abstraction
- **Testing Framework**: 600 lines - Comprehensive test suite

The system uses Sled as its storage backend, providing crash recovery and ACID guarantees. Network communication supports both HTTP REST APIs and custom TCP protocols for efficient bulk operations.

## 10 Availability

FoldDB is available as open-source software under the MIT license. The complete source code, documentation, and example applications are available at:

`https://github.com/datafold/folddb`

The project includes comprehensive documentation, API references, and deployment guides to facilitate adoption and contribution from the research community.

## References


[1] R. Hickey, "The Datomic Information Model," in *Proceedings of the 2012 European Lisp Symposium*, 2012, pp. 17-24.

[2] EventStore Team, "Event Store Documentation," EventStore Ltd., 2024. [Online]. Available: https://eventstore.com/docs/

[3] Apache Software Foundation, "Apache Avro Specification," 2015. [Online]. Available: https://avro.apache.org/docs/current/spec.html

[4] Apache Software Foundation, "Apache Parquet Documentation," 2013. [Online]. Available: https://parquet.apache.org/docs/

[5] Oracle Corporation, "Database VLDB and Partitioning Guide," Oracle Database Documentation, 2019.

[6] P. A. Bernstein and V. Hadzilacos, *Concurrency Control and Recovery in Database Systems*. Boston: Addison-Wesley, 1987.

[7] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco: Morgan Kaufmann, 1993.

[8] M. Stonebraker et al., "C-Store: A Column-oriented DBMS," in *Proceedings of the 31st VLDB Conference*, 2005, pp. 553-564.

[9] D. J. DeWitt et al., "GAMMA - A High Performance Dataflow Database Machine," in *Proceedings of the 12th VLDB Conference*, 1986, pp. 228-237.

[10] S. Klabnik and C. Nichols, *The Rust Programming Language*. San Francisco: No Starch Press, 2018.