Vantage – A Java Architect's Perspective

As a battle-hardened Java architect with 15+ years in enterprise trenches—wrangling monolithic Spring Boot apps, Hibernate's endless annotations, and the fragility of JPA across diverse databases like Postgres, MongoDB, and Oracle—Vantage has captivated my interest. I've seen too many "universal" persistence solutions crumble under real-world complexity: overpromised ACID guarantees leading to XA failures and downtime, rigid universal validations choking on legacy schemas, migration marathons with Liquibase breaking backward compatibility, and dialect-specific hacks bloating codebases.

Vantage, developed by Romans Malinovskis and inspired by OpenDAL's pluggable architecture, takes a refreshingly different approach. This Rust-based framework (version 0.3 released, 0.4 in development) isn't trying to be another ORM—it's a pragmatic toolkit for type-safe, backend-agnostic persistence. Rather than fighting against database differences, it embraces them through specialized crates like `vantage-surrealdb`, `vantage-mongodb`, and `surreal-client`, each providing deep integration while maintaining unified abstractions.

What draws me to Vantage is its "realistic pragmatism": it emphasizes idempotency for retryable operations, contextual validation to handle dirty data gracefully, schema absorption without forcing migrations, and compile-time safety via Rust's trait system. You can evolve systems gradually without ripping up existing code. With planned Python/Java bindings in 0.4, it could become a seamless bridge between Rust's performance and familiar ecosystems. Let's explore this layer by layer, contrasting with the Java pain points I've endured.

Foundation: Types and Datasets – Uniform Abstractions Over Diverse Backends

Vantage's architecture separates concerns elegantly through specialized crates. The `vantage-types` crate provides Entity definitions and Record conversion traits that enable

seamless data flow between strongly-typed Rust structs and storage values (JSON, CBOR, etc.). Unlike Hibernate's annotation hell where adding a simple field requires updating mappers, validators, and DTOs, Vantage entities work naturally with serde for automatic serialization.

The `vantage-dataset` crate defines fundamental capabilities through traits: `ReadableDataSet` for read-only sources (CSV, APIs), `InsertableDataSet` for append-only scenarios (message queues, event streams), and `WritableDataSet` for full CRUD operations. These traits form the "absolute minimum" contracts any storage must honor, enabling uniform access across diverse backends—unlike Spring Data's fragmented ecosystem where switching from JPA to reactive MongoDB means rewriting your entire repository layer.

For illustration, a basic entity and dataset setup:

```
use vantage_dataset::traits::{DataSet, WritableDataSet,
ReadableDataSet};
use vantage_types::Entity;
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, Clone)]
struct User {
    name: String,
    email: String,
}

// Any storage that implements WritableDataSet<User>
async fn demo_crud<T>(ds: &T) -> Result<(), Box<dyn
std::error::Error>>
where
    T: WritableDataSet<User>,
{
    let user = User { name: "Alice".to_string(), email:
"alice@example.com".to_string() };

    // Idempotent operations - safe to retry
```

```
    let saved = ds.insert(&"user-123".to_string(), &user).await?;
    ds.replace(&"user-123".to_string(), &saved).await?;
    ds.delete(&"user-123".to_string()).await?; // Succeeds even if
already deleted

    Ok(())
}
```

This uniformity prevents the architectural rewrites I've endured in Java when migrating from PostgreSQL to MongoDB—Vantage abstracts the differences while preserving each backend's strengths.

Expressions: Composable, Delayed Queries with Type-Safe Power

The `vantage-expressions` crate provides SQL-injection-safe query building through the `expr!` macro and composable expression system. Unlike JPA's Criteria API or QueryDSL, which tangle in metamodels and runtime type mismatches, Vantage expressions separate template from parameters, building synchronously but executing asynchronously to prevent N+1 query disasters common in Java stacks.

The core innovation is template-based composition with delayed execution:

```
use vantage_expressions::expr;

// Build conditions separately - no database hits
let age_condition = expr!("age > {} AND status = {}", 21, "active");
let final_query = expr!("SELECT * FROM users WHERE {}",
(age_condition));
```

Parameters stay separate from templates, preventing SQL injection while maintaining readability. Unlike PreparedStatements in JDBC where you lose query structure, expressions preserve the logical flow.

For cross-database scenarios, `DeferredFn` enables async operations within sync query building:

```
// API call wrapped in deferred function
async fn get_active_user_ids() -> Result<serde_json::Value> {
    Ok(serde_json::json!([1, 2, 3, 4, 5]))
}

// Build query synchronously - API call deferred until execution
let query = expr!(
    "SELECT * FROM orders WHERE user_id = ANY({})",
    { DeferredFn::from_fn(get_active_user_ids) }
);

// Execute - API call happens automatically during execution
let orders = db.execute(&query).await?;
```

This eliminates the async propagation nightmare where one external dependency forces your entire query building pipeline to become async, breaking existing synchronous APIs.

Tables: High-Level Entity Management with Unified CRUD Operations

In practice, implementing unique DataSet traits for every entity type creates excessive boilerplate. Vantage solves this through the generic `Table<DataSource, Entity>` struct, which automatically implements all necessary dataset traits while providing schema-aware field access and column support. This eliminates the repository pattern explosion common in Spring Data, where switching between JPA and MongoDB requires entirely different abstractions.

Tables provide a unified interface that adapts to datasource capabilities—enabling CRUD operations when supported, field-based conditions through indexing syntax, and relationship navigation without N+1 queries. Unlike Hibernate's rigid entity mappings, tables absorb schema evolution gracefully, supporting column additions without migration scripts or code changes.

Here's how table-based entity management works:

```
use vantage_table::Table;
use vantage_surrealdb::SurrealDB;
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, Clone)]
struct Client {
    name: String,
    email: String,
    is_paying_client: bool,
}

impl Client {
    fn table(db: SurrealDB) -> Table<SurrealDB, Client> {
        Table::new("client", db)
            .with_id_column("id")
            .with_column_of::<String>("name")
            .with_column_of::<String>("email")
            .with_column_of::<bool>("is_paying_client")
            .into_entity()
    }
}

// Field-based conditions using indexing syntax
let clients = Client::table(surrealdb());
let paying_clients =
clients.with_condition(clients["is_paying_client"].eq(true));

// Execute unified operations regardless of backend
let results = paying_clients.get().await?;
let count = paying_clients.get_count().await?;
```

This approach provides type-safe field access, automatic serialization, and consistent APIs across different storage backends—eliminating the fragmentation that plagues Java persistence layers.

Vendor-Specific Capabilities and Method Triads

Table capabilities expand based on the underlying datasource traits. Basic `TableSource` provides fundamental CRUD operations, while `QuerySource` adds expression-building methods and `SelectSource` enables advanced query modification. Vantage delegates these implementations to vendor-specific crates like `vantage-surrealdb`, ensuring optimal performance for each database while maintaining unified APIs.

`QuerySource` extends tables with `_expr` methods that return expressions instead of executing immediately—for instance, alongside `async count()`, you get `count_expr()` returning a counting expression for composition. `SelectSource` adds `_query` methods returning modifiable query builders, enabling fine-tuned control before execution.

The power emerges in method triads offering immediate, expressive, and query-builder approaches:

```
// Count operations - three patterns
async fn get_count() -> i64;                    // Execute immediately
fn get_count_query() -> Expression;             // Return expression
for composition
fn select().as_count() -> SurrealReturn;        // Query builder then
execute

// Sum operations - three patterns
async fn get_sum(field) -> T;                   // Execute immediately
fn get_sum_query(field) -> Expression;          // Return expression
for composition
fn select().as_sum(field) -> SurrealReturn;     // Query builder then
execute
```

The `select()` method returns vendor-specific query builders— `SurrealSelect` for SurrealDB, `MySqlSelect` for MySQL, etc.—each exposing their database's full query capabilities while maintaining the unified pattern.

Each approach serves distinct purposes:

- **Immediate execution** (`get_count()`) provides the lowest common denominator—maximum compatibility across all datasources, ideal for simple operations
- **Expression building** (`get_count_query()`) enables composition and encapsulation—embed counts within conditions or nest into larger expressions
- **Select builders** (`select().as_count()`) unlock full vendor capabilities—leverage database-specific optimizations, advanced syntax, and complex aggregations

The ability to extract vendor-specific select builders proves particularly valuable for advanced aggregation patterns, custom reporting pipelines, and complex analytical queries—features planned for enhancement in future Vantage versions.

Expressions remain unflattened until execution, allowing cross-database queries with automatic type mapping. When `fetch()` finally executes, Vantage intelligently embeds same-database queries while handling cross-database operations seamlessly—avoiding the complexity of distributed transactions.

Advanced Table Operations: Joins, Transactions, and Reliability Patterns

High-level aggregation isn't the only feature that builds upon entity tables. Vantage provides additional sophisticated operations including relationship joins and transactional operations that intercept and coordinate table operations for ACID guarantees.

Joins operate at two levels: intra-database joins leverage native database capabilities for efficiency, while cross-database joins use independent queries with intelligent reconstruction. Operations prioritize idempotency over Java's brittle XA transactions, which frequently deadlock heterogeneous database systems in production environments.

Transactions use scoped, trait-based patterns that encourage idempotent designs suitable for cross-database retries rather than false ACID promises. This avoids JTA's configuration complexity while providing automatic rollback on errors through Rust's drop semantics and uniform error handling via vantage-core's Result types.

Planned transaction example for 0.4:

```
let tx = Transaction::start();
accounts.with_transaction(tx).map(|a| {
    a.balance -= 10;
    Ok(a)
}).await?;
tx.commit().await?;
```

Idempotency resolves partial failures pragmatically, unlike Spring's @Transactional partial commits.

ActiveRecord: Entity Modification with Change Tracking

Vantage provides a simple but effective ActiveRecord pattern through `ActiveEntity<D, E>`, which wraps entities with their IDs and dataset references for safe mutations. Unlike Hibernate's complex lazy loading and cascade configurations, Vantage's approach focuses on straightforward field modification with explicit save operations.

The ActiveRecord implementation leverages Rust's `Deref` / `DerefMut` traits, allowing direct field access on the wrapped entity while tracking changes for efficient persistence:

```
// Get an entity wrapped for modification
let mut client_record =
clients.get_record(&client_id).await?.unwrap();

// Modify fields directly through Deref
client_record.name = "Updated Name".to_string();
client_record.is_paying_client = true;

// Explicit save - uses replace() for idempotent updates
client_record.save().await?;
```

This approach avoids Bean Validation's universal schema constraints and cascade complexity, providing predictable behavior where data can change externally without breaking application assumptions.

Relationships: Navigation Without Cascade Complexity

Table relationships in Vantage use the `get_ref_as()` method to navigate between related entities without Hibernate's cascade pitfalls. Relationships are defined during table construction and accessed through type-safe downcasting:

```
// Define relationships in table setup
impl Client {
    fn table(db: SurrealDB) -> Table<SurrealDB, Client> {
        Table::new("client", db)
            .with_column_of::<String>("name")
            .with_one("bakery", "bakery", move ||
Bakery::table(db2.clone()))
            .with_many("orders", "client", move ||
Order::table(db3.clone()))
    }
}

// Navigate relationships using trait methods
let clients = Client::table(db);
let bakery = clients.get_ref_as::<SurrealDB, Bakery>("bakery")?;
let orders = clients.get_ref_as::<SurrealDB, Order>("orders")?;

// Or use convenient syntactic sugar methods
let orders = clients.ref_orders();  // Type-safe, no downcasting
needed
let bakery = clients.ref_bakery();
```

This provides relationship traversal without lazy loading complexity or cascade management, maintaining explicit control over when and how related data is accessed.

Version 0.4 will enhance this further by enabling traversal between `ActiveEntity` and `Table`, offering developers a choice between traversin into "single-record-dataset" or async loading of ActiveEntity.

Calculated Columns: Sub-Query Aggregations and Expression Fields

Calculated columns represent one of Vantage's most powerful departures from traditional ORM patterns. Unlike JPA's @Formula annotations, which require raw SQL and break database portability, or Hibernate's calculated properties that force application-level computation, Vantage calculated columns leverage the expression system to generate optimal vendor-specific subqueries.

Relationship aggregation demonstrates the elegance—account balances calculated from transaction sums:

```
impl Account {
    fn table(db: SurrealDB) -> Table<SurrealDB, Account> {
        Table::new("account", db)
            .with_column_of::<String>("name")
            .with_many("transactions", "account_id", |a|
Transaction::table(a.db().clone()))
            .with_expression("balance", |t| {
                let tr = t.link_transactions();
                tr.sum(tr.total()).expr()
            })
    }

    // Type-safe helper for relationship access
    fn link_transactions(&self) -> Table<Transaction, SurrealDB> {
        self.get_link("transactions").into_entity()
    }
}
```

What makes this particularly compelling is nested composition—if `transaction.total()` itself is a calculated column (perhaps `quantity * unit_price + tax`), Vantage transparently handles the expression nesting. Unlike Spring's @Transient computations that force N+1 queries, these calculations happen entirely at the database level with optimal performance.

Direct expression fields handle complex business logic:

```
Account::table(db)
    .with_expression("total_vat", |t| t.get_expr("{vat_rate} *
{total}"))
    .with_expression("profit_margin", |t| t.get_expr("({revenue} -
{cost}) / {revenue}"))
```

This flexibility proves invaluable during refactoring—you can pivot between stored and calculated columns as performance optimization dictates, without changing consuming code. A calculated field today becomes a materialized column tomorrow simply by removing the expression and adding migration logic, avoiding the architectural rewrites common when switching between JPA entity properties and database views.

Testing with Mocks: Lightweight, Comprehensive Coverage

Vantage provides sophisticated mocking capabilities that far exceed Testcontainers' heavyweight approach or Mockito's fragile stubs. The `MockBuilder` pattern matches queries precisely while the `MockTableSource` simulates complete table operations with in-memory data, enabling fast CI builds without external dependencies.

Cross-database expression mocking with nested queries:

```
use vantage_expressions::{expr, mocks::mockbuilder,
traits::expressive::DeferredFn};
use serde_json::json;

// API call that fetches user IDs asynchronously
async fn get_active_user_ids() ->
vantage_core::Result<serde_json::Value> {
    Ok(json!([1, 2, 3, 4, 5]))
}

let mock = mockbuilder::new()
```

```
        .with_flattening()
        .on_exact_select(
            "SELECT * FROM orders WHERE user_id IN (SELECT id FROM users
    WHERE department = \"engineering\")",
            json!([{"id": 100, "user_id": 1, "total": 250.00}])
        )
        .on_exact_select(
            "SELECT * FROM orders WHERE user_id = ANY([1,2,3,4,5])",
            json!([{"id": 101, "user_id": 2, "total": 150.00}])
        );

    // Nested subquery - flattened automatically
    let user_subquery = expr!("SELECT id FROM users WHERE department =
    {}", "engineering");
    let orders_query = expr!("SELECT * FROM orders WHERE user_id IN ({})",
    (user_subquery));

    // Deferred API call embedded in query
    let api_query = expr!("SELECT * FROM orders WHERE user_id = ANY({})",
    {
        DeferredFn::from_fn(get_active_user_ids)
    });

    let orders = mock.execute(&orders_query).await?;
    assert_eq!(orders.as_array().unwrap()[0]["total"], 250.00);
```

Table-level mocking for integration testing:

```rust
use vantage_table::mocks::MockTableSource;

let mock_db = MockTableSource::new()
    .with_data("clients", vec![
        json!({"id": "1", "name": "Alice", "is_paying": true}),
        json!({"id": "2", "name": "Bob", "is_paying": false}),
    ]);

let clients = Client::table(mock_db);
let count = clients.get_count().await?;
assert_eq!(count, 2);
```

This approach exercises real code paths including expression building, query generation, and result deserialization—providing comprehensive coverage without the brittleness of Java's mocking frameworks that break on refactoring.

The mock system's trait-based architecture enables sophisticated third-party integrations. Performance-critical scenarios can implement custom `QuerySource` mocks using specialized engines like `MockSurrealEngine` with exact parameter matching, or integrate with property-based testing frameworks like Proptest for generating realistic dataset variations. Unlike JUnit's rigid mocking that requires deep framework knowledge, Vantage's mocks simply implement standard traits, allowing seamless integration with any Rust testing ecosystem while maintaining the same high-performance characteristics as production datasources.

Future Capabilities: TableRouter, Live Sync

TableRouter (planned for 0.4) will provide operation-level routing with ACL enforcement, avoiding the gateway complexity common in Java microservice architectures. Unlike Spring Cloud Gateway's configuration overhead, TableRouter integrates directly into table operations:

```rust
use vantage_table::{TableRouter, Table};
use vantage_kafka::KafkaSink;
```

```
let primary_table = User::table(postgres());
let audit_sink = KafkaSink::new("user-edits-topic");

let routed_table = TableRouter::new(primary_table)
    .route_read(|op| op.to(primary_table))
    .route_edit(|op| op.to(audit_sink).fallback(primary_table))
    .with_acl(|user, op, record| {
        if op.is_edit() && record["role"] == "admin" {
            Err("Admins cannot be edited directly".into())
        } else {
            Ok(())
        }
    });

// Use like any table - routing happens transparently
let vip_users =
routed_table.with_condition(routed_table["role"].eq("vip"));
vip_users.map(|u| { u.status = "active"; u }).await?; // Routes
through ACL → Kafka
```

Live Tables via `vantage-live` address the responsive UI problem that plagues enterprise Java applications. Unlike JPA's optimistic locking failures, Live Tables provide conflict-aware editing sessions with async persistence:

```
use vantage_live::LiveTable;

// Backend + cache architecture
let backend = Bakery::table(surrealdb());
let cache = ImTable::new(&im_ds, "bakery_cache");
let live_table = LiveTable::new(backend, cache).await?;

// Edit session with change tracking
let mut edit = live_table.edit_record("bakery:123").await?;
edit.name = "Updated Name".to_string();
edit.profit_margin = 0.25;
```

```
// Field-level diffing for UI highlighting
for field in edit.get_modified_fields() {
    highlight_field_in_ui(&field);
}

// Async save with conflict detection
match edit.save().await? {
    SaveResult::Saved => update_ui_success(),
    SaveResult::Error(e) => show_validation_errors(&e),
    SaveResult::PartialSave(fields) => handle_conflicts(&fields),
}
```

These capabilities demonstrate Vantage's architectural philosophy: **idempotency over illusions, absorption over migrations, traits over configuration**. This pragmatic approach avoids the technical debt accumulation endemic to Java enterprise stacks, fostering systems that evolve gracefully rather than requiring periodic rewrites.

Architectural Foundations: Beyond the API Surface

Several foundational design decisions distinguish Vantage from Java persistence frameworks, addressing pain points that often surface only after extended production use.

Uniform Error Handling via `vantage-core` wraps diverse persistence SDK errors into consistent types, eliminating the error model fragmentation common in Java where JDBC SQLException, Spring's DataAccessException, and Elasticsearch's ResponseException each require different handling patterns:

```
// All persistence operations return uniform VantageError
let result: Result<User, VantageError> = users.get("123").await;
match result {
    Ok(user) => process(user),
    Err(e) => log_error(&e), // Same error type regardless of backend
}
```

Expression Encapsulation uses three primitives—Value (typed literals), Nested (same-DB subqueries), and Deferred (cross-DB promises)—enabling transparent composition without

the async propagation nightmares that plague Java's reactive streams when mixing database and API calls.

Runtime Column Metadata through configurable flags (HashMap enums like "hidden", "system") drives generic UI behavior, supporting multiple table configurations per entity (`User::admin_table()` vs `User::public_api_table()`). This flexibility avoids Spring Data's rigid @Entity constraints that force separate classes for different access patterns.

Reference Navigation Semantics distinguish `get_ref()` (independent datasets, cross-DB capable via deferred expressions) from `get_link()` (correlated subqueries for calculated columns, same-DB only). This architectural choice enables seamless cross-database relationships without the JOIN limitations that constrain JPA across schemas.

ActiveEntity Lifecycle Management uses `NewActiveEntity<E, DS>` for creation flows, leveraging Rust's type system to prevent invalid operations (like saving before entity creation) that often require runtime validation in Java builders or factory patterns.

These design patterns reflect Vantage's core philosophy of **realistic pragmatism**—acknowledging that enterprise data is messy, systems evolve incrementally, and type safety should prevent common mistakes without imposing artificial constraints. It's a refreshing departure from Java's tendency toward comprehensive frameworks that promise universal solutions but break down under real-world complexity.

Language bindings

Looking ahead, Vantage plans language bindings for Java and Python ecosystems, enabling integration with existing enterprise infrastructure without abandoning current technology investments. Early Python bindings in `example_python` demonstrate async table operations through PyO3, while planned Java bindings will provide native JVM access to Vantage's capabilities. This positions Vantage not as a replacement for existing systems, but as a bridge that brings Rust's performance and safety benefits to familiar development environments, allowing teams to adopt incrementally rather than requiring full rewrites.