**DEnode DataBase-schema optimization-tool**

**Alex Osterneck, CLA, MSCS // ai70000 Ltd.**

**April 15th, 2025**

**( AI-automation for: optimizing Denormalization / Normalization of any database )**

**PROOF-OF-CONCEPT & PROTOTYPE & SOURCE-CODE**

## I. CURRENT STATE of DATABASE-OPTIMIZATION for NORMALIZATION / DE-NORMALIZATION

### 1. Database Performance Monitoring Tools

These identify slow queries, redundant joins, and data access patterns:

**SolarWinds Database Performance Analyzer, Redgate SQL Monitor, New Relic for Databases, Datadog Database Monitoring**

(These help **infer when denormalization might help** (e.g., repeated joins, excessive latency).

---

### 2. Query Optimizers / Analyzers

**PostgreSQL EXPLAIN/ANALYZE, SQL Server Query Store, MySQL Workbench Performance Reports**

(Not off-the-shelf decisions, but they provide **raw data to justify schema changes**.)

---

### 3. Workload & Schema Analysis Tools

**Vertabelo**, **ER/Studio**, **Toad Data Modeler**

Some tools **model and simulate schema variations** (e.g., normalized vs. flattened), letting you compare expected outcomes manually.

---

### 4. Custom Heuristics in Data Warehousing Platforms

**Snowflake**, **BigQuery**, **Amazon Redshift**: These platforms often **recommend denormalized schemas** due to distributed architecture, and their performance analyzers may help justify structure changes.

---

## II. WHAT DOESN'T EXIST

No tool currently:

Automatically models all normalization/denormalization tradeoffs

Benchmarks each structure

Gives a final recommendation like: "Denormalize table X and Y for 35% faster joins"

---

Current industry-standard workaround:

Custom scripts + logs + BI dashboards (e.g., Looker, Metabase) + query stats can create a **feedback loop** that only approximates optimization.

## III. INTRODUCING: DEnode DataBase-schema optimization-tool

**By: Alex Osterneck, CLA, MSCS // ai70000 Ltd.**

**(new software for optimization of normalization / denormalization in databases)**

*step-by-step engineering blueprint of **fully automated product** that analyzes large legacy databases and intelligently recommends and applies normalization/denormalization changes to improve performance.*

---

## DEnode Tech-Stack

**1. Backend**
- Python 3.11+: Core programming language
- Flask: Web framework for the web interface
- SQLAlchemy: ORM and database abstraction
- Gunicorn: WSGI HTTP server for production deployment
- Click: Command-line interface framework

**2. Database Drivers**
- SQLite: Built-in database for local development
- psycopg2: PostgreSQL connector for production databases
- SQLAlchemy Dialects: Support for multiple database types

**3. Performance Analysis**
- SQL Query Log Parser: Custom parser for SQL query analysis
- EXPLAIN Analyzer: Executes and interprets EXPLAIN/EXPLAIN ANALYZE output
- Schema Introspection: Extracts database schema metadata
- Benchmarking Engine: Measures query performance under various conditions
- Concurrency Testing: Simulates load with parallel query execution

**4. Storage**
- Metadata Store: Hybrid JSON/SQLite storage system for analysis results
- Versioning System: Maintains history of schema snapshots and recommendations
- Session Storage: Web interface state management
- Database Name Fuzzy Matching: Smart database lookup with normalization

**5. Interface**
- Web UI: Bootstrap-based responsive interface
- CLI: Command-line tools for automation
- Web API: REST endpoints for integration
- Interactive Visualizations: Schema and relationship visualization
- Error Handling: Comprehensive error display and recovery

**6. Optional ML (Future Enhancement)**
- Query Pattern Recognition: Machine learning pattern detection in query logs
- Predictive Performance Analysis: Forecast performance under load
- Recommendation Confidence Scoring: Enhanced reliability of recommendations
- Automatic Index Optimization: Self-tuning database configurations
- Anomaly Detection: Identification of unusual query patterns

# DEnode System-Design

## 1. Modular Architecture
- Core Engine: Central optimization logic independent of interface
- Pluggable Components: Swappable analyzers, storage backends, and interfaces
- Configuration System: Environment-based with sensible defaults
- Dependency Injection: Components receive dependencies vs. creating them
- Service Layer: Clear separation between business logic and technical concerns

## 2. Data Flow
- Input Layer: Database connection details and query logs
- Extraction Phase: Schema metadata collection from target database
- Analysis Phase: Query pattern identification and statistical analysis
- Recommendation Engine: Heuristic-based optimization suggestions
- Implementation Phase: SQL generation for schema modifications
- Validation Phase: Before/after performance testing

## 3. Processing Pipeline
- Connection Handler: Manages database connections and credentials
- Schema Extractor: Builds comprehensive table/column/constraint model
- Query Parser: Tokenizes and classifies SQL query patterns
- Pattern Analyzer: Identifies usage patterns and access frequencies
- Heuristic Engine: Applies optimization rules to schema and query data
- Plan Generator: Creates SQL implementation scripts for recommendations
- Benchmarking System: Measures performance impact of changes

## 4. Storage Architecture
- Metadata Database: SQLite for analysis results and recommendations
- File System: JSON storage for schemas and configuration
- In-Memory Cache: Performance optimization for repeated access
- Version Control: Historical tracking of schema changes
- Persistence Layer: Abstract storage interface for multiple backends

## 5. Interface Design
- MVC Architecture: Model-View-Controller pattern for web application
- REST API: Resource-based interface for programmatic access
- CLI Interface: Command-line tools for scripting and automation
- Template System: Jinja2 templates for UI rendering
- Component Library: Bootstrap-based responsive design system

## 6. Security Model
- Connection Isolation: Database credentials never stored long-term
- Session Management: Short-lived web sessions with secure tokens
- Authentication: Optional user authentication for sensitive operations
- Access Control: Role-based permissions for multi-user deployments
- Data Sanitization: Input validation and SQL injection prevention

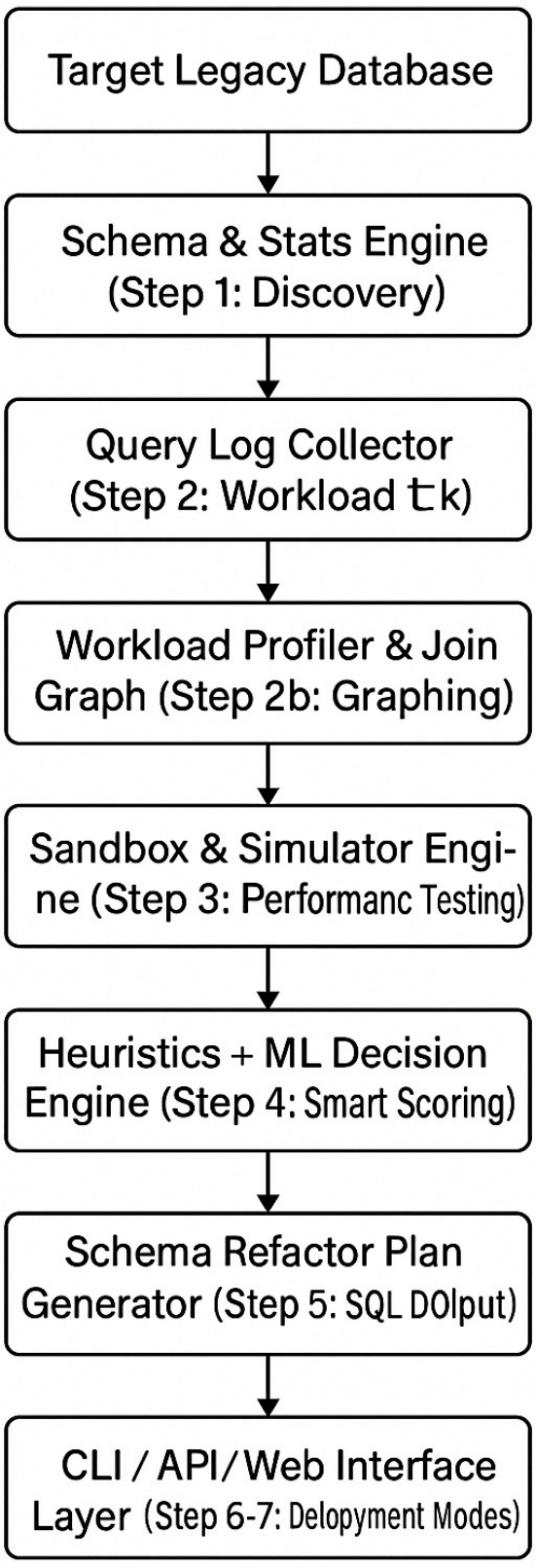
## 7. Deployment Architecture
- Containerization: Docker support for consistent environments
- Configuration Management: Environment variables and config files
- Stateless Design: Multiple instances can run in parallel
- Resource Management: Configurable resource limits for intensive operations
- Health Monitoring: Status endpoints and diagnostic tools

---

---

---

## SYSTEM ARCHITECTURE DIAGRAM



## CODE SCAFFOLDING FOR ALL STEPS

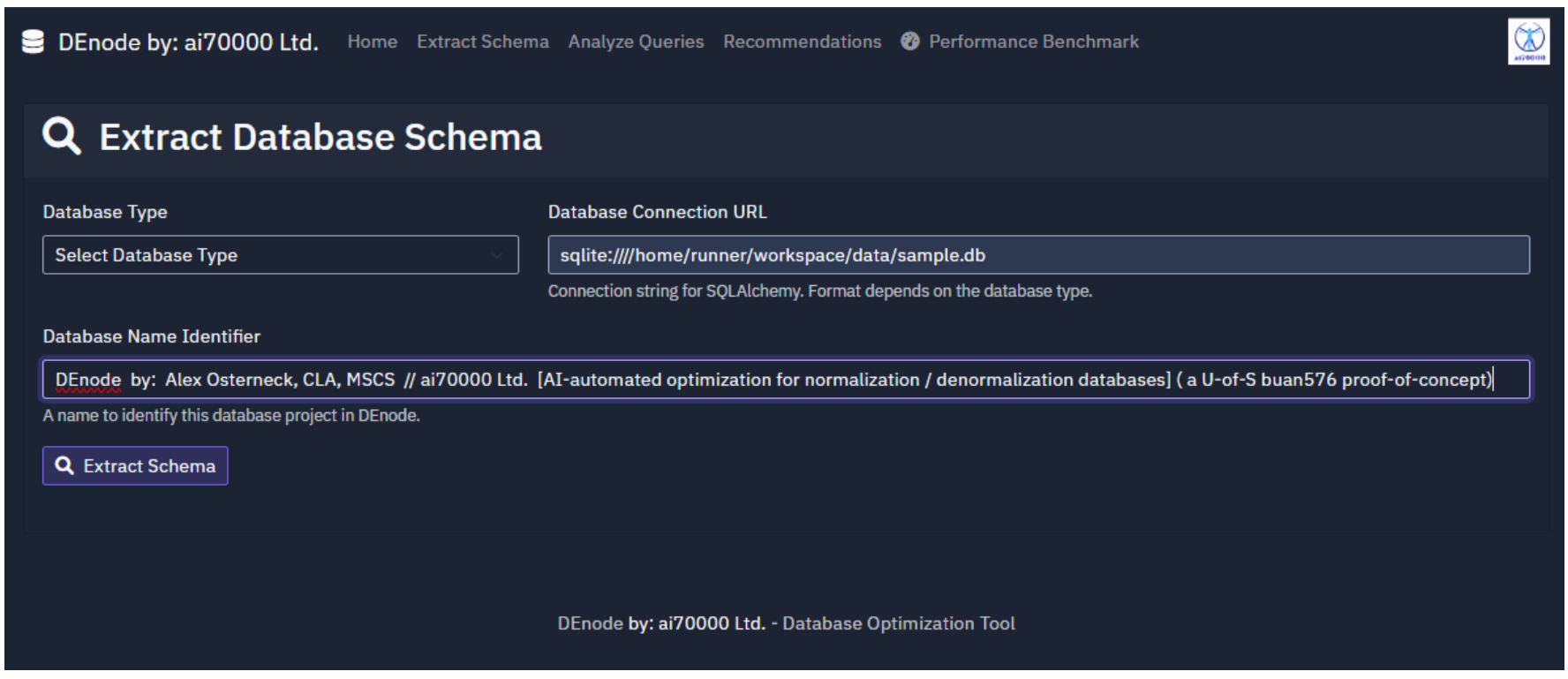
```
denode/
├── main.py                          # Main entry point for both CLI and web interface
├── config.py                        # Configuration handling with environment variables
├── db/
│   ├── __init__.py                  # Package initialization
│   ├── connector.py                 # Database connection management
│   ├── schema_extractor.py          # Extracts database schema information using SQLAlchemy
│   ├── query_log_analyzer.py        # Parses and analyzes query logs for patterns
│   └── performance_metrics.py       # Collects database performance metrics using EXPLAIN
├── engine/
│   ├── __init__.py                  # Package initialization
│   ├── simulator.py                 # Simulates performance with EXPLAIN ANALYZE
│   ├── heuristics.py                # Contains optimization recommendation rules
│   ├── plan_generator.py            # Generates SQL implementation plans
│   └── benchmark.py                 # Performance benchmarking utilities
├── storage/
│   ├── __init__.py                  # Package initialization
│   ├── metadata_store.py            # Stores analysis data in SQLite with JSON serialization
│   └── session.py                   # Web session management for Flask application
├── ui/
│   ├── __init__.py                  # Package initialization
│   ├── cli.py                       # Command-line interface using Click
│   └── helpers.py                   # UI helper functions for formatting output
├── webapp/
│   ├── __init__.py                  # Package initialization
│   ├── app.py                       # Flask web application
│   ├── routes.py                    # Web route handlers
│   ├── helpers.py                   # Web helper utilities
│   ├── templates/                   # Jinja2 HTML templates
│   │   ├── base.html                # Base template with common layout
│   │   ├── index.html               # Home page
│   │   ├── extract.html             # Schema extraction page
│   │   ├── analyze.html             # Query analysis page
│   │   ├── recommendations.html     # Recommendations display page
│   │   ├── sql_plan.html            # SQL implementation plan page
│   │   └── error.html               # Error page
│   └── static/                      # Static web assets
│       ├── css/                     # CSS stylesheets
│       ├── js/                      # JavaScript files
│       └── img/                     # Image assets
├── utils/
│   ├── __init__.py                  # Package initialization
│   ├── profiler.py                  # Performance profiling tools
│   ├── logger.py                    # Logging configuration
│   └── url_helpers.py               # URL encoding/decoding helpers for special characters
├── metadata/                        # Storage directory for metadata
│   └── metadata.db                  # SQLite database for metadata storage
├── data/                            # Data directory for sample databases
│   └── sample.db                    # Sample SQLite database for testing
├── tests/                           # Test directory
│   ├── __init__.py                  # Package initialization
│   ├── test_schema_extractor.py     # Tests for schema extraction
│   ├── test_query_analyzer.py       # Tests for query log analysis
│   └── test_heuristics.py           # Tests for recommendation engine
├── pyproject.toml                   # Project dependencies and metadata
├── README.md                        # Project documentation
└── CHANGELOG.md                     # Version history and changes
```

## IV. WORKING PROTOTYPE

## ALLOWS INPUT OF YOUR DATABASE-url FOR AUTO-OPTIMIZATION via DEnode APP

## AUTO-GENERATION of all SQL-commands for Normalization, Denormalization, & Indexing

## DEnode Database Schema Optimization Tool by: Alex Osterneck, CLA, MSCS // ai70000, Ltd.

DEnode Home Extract Schema Analyze Queries Recommendations

## Extract Database Schema

Database Type: SQLite

Database Connection URL: sqlite:///path/to/database.db

Connection string for SQLAlchemy. Format depends on the database type.

Database Name Identifier: DEnode by: Alec Osterneck, CIA, NSCS// ai70000 Ltd. [AI automation to optimize normalization / denormalization for any database] (a U-of-S huan576 proof-of-concept)

A name to identify this database project in DEnode.

Extract Schema

## Schema Extraction Results

### Schema Overview

Successfully extracted schema with 4 tables.

| Table | Columns | Primary Key | Foreign Keys | Indexes | Details |
|---|---|---|---|---|---|
| customers | 11 | Yes | 0 | 0 | Hide Columns |
| ↳ customer_id | INTEGER PK | | | | |
| ↳ customer_name | TEXT NOT NULL | | | | |
| ↳ customer_email | TEXT | | | | |
| ↳ customer_phone | TEXT | | | | |
| ↳ customer_address | TEXT | | | | |
| ↳ customer_city | TEXT | | | | |
| ↳ customer_state | TEXT | | | | |
| ↳ customer_zipcode | TEXT | | | | |
| ↳ customer_country | TEXT | | | | |
| ↳ date_registered | DATE | | | | |
| ↳ last_login | TIMESTAMP | | | | |
| order_items | 6 | Yes | 2 | 1 | Show Columns |

## Extract Database Schema

Database Type: Select Database Type

Database Connection URL: e.g., postgresql://username:password@localhost:5432/database

Connection string for SQLAlchemy. Format depends on the database type.

Database Name Identifier: e.g., my_project_db

A name to identify this database project in DEnode.

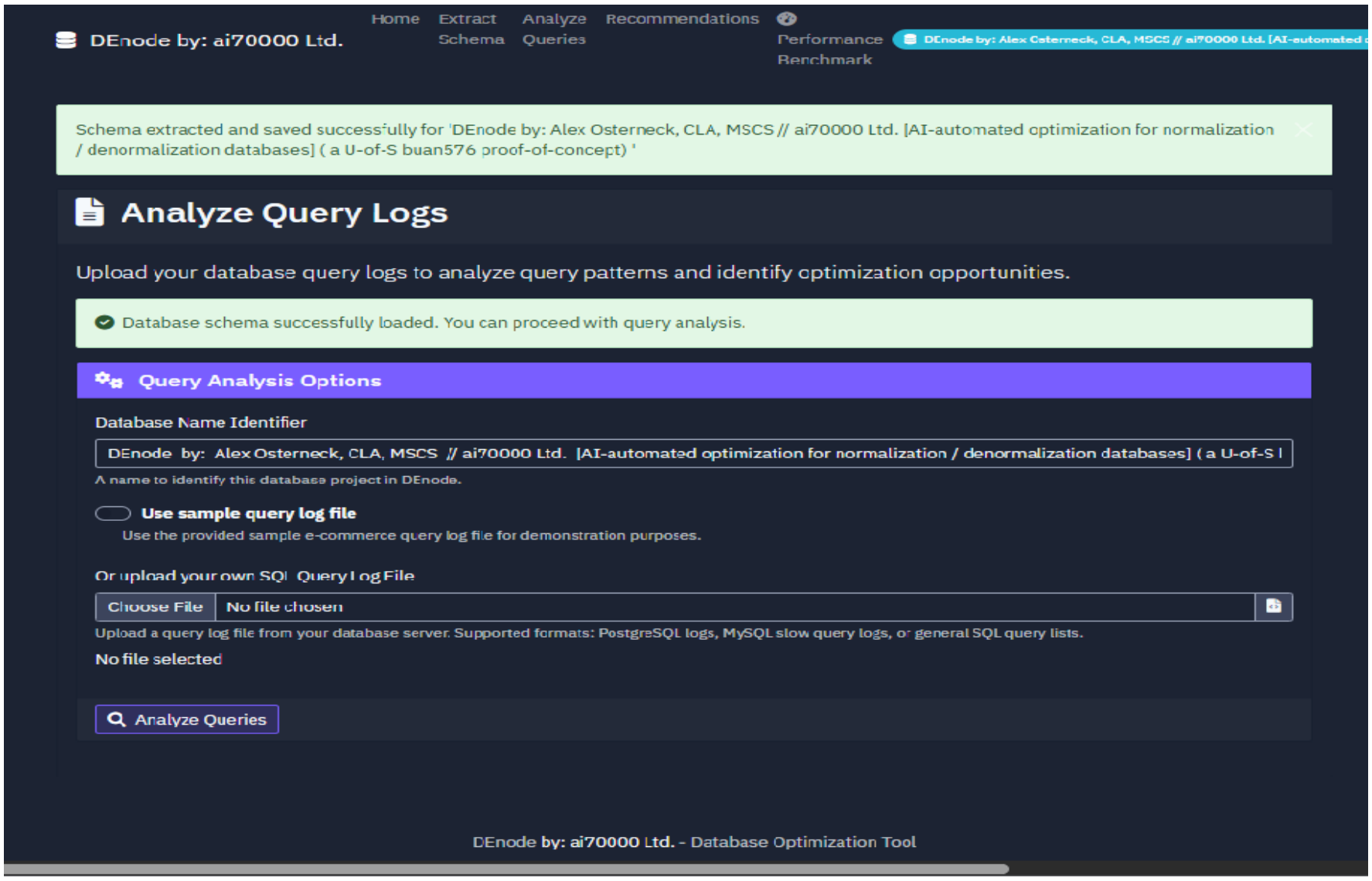
Extract Schema

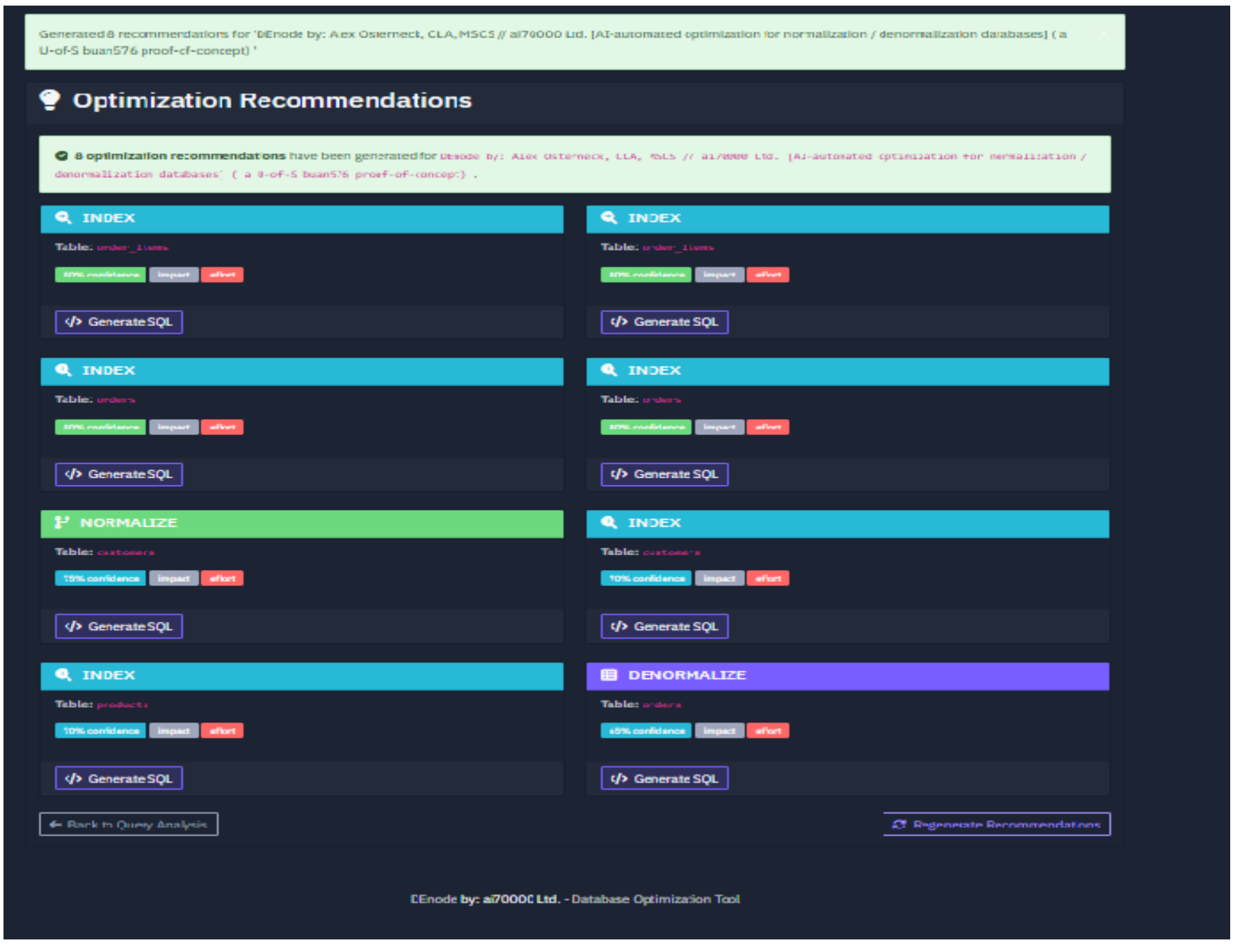
## Schema Extraction Results

### Schema Overview

Successfully extracted schema with 4 tables.

| Table | Columns | Primary Key | Foreign Keys | Indexes | Details |
|---|---|---|---|---|---|
| customers | 11 | Yes | 0 | 0 | Show Columns |
| order_items | 6 | Yes | 2 | 1 | Show Columns |
| orders | 11 | Yes | 1 | 2 | Hide Columns |
| ↳ order_id | INTEGER PK | | | | |
| ↳ customer_id | INTEGER NOT NULL | | | | |
| ↳ order_date | TIMESTAMP | | | | |
| ↳ order_status | TEXT | | | | |
| ↳ shipping_address | TEXT | | | | |
| ↳ shipping_city | TEXT | | | | |
| ↳ shipping_state | TEXT | | | | |
| ↳ shipping_zipcode | TEXT | | | | |
| ↳ shipping_country | TEXT | | | | |
| ↳ payment_method | TEXT | | | | |


DEnode by: ai70000 Ltd.
Home
Extract Schema
Analyze Queries
Recommendations
Performance Benchmark
DEnode by: Alex Osterneck, CLA, MSCS // ai70000 Ltd. [AI-automated
Schema extracted and saved successfully for 'DEnode by: Alex Osterneck, CLA, MSCS // ai70000 Ltd. [AI-automated optimization for normalization / denormalization databases] ( a U-of-S buan576 proof-of-concept) '
Analyze Query Logs
Upload your database query logs to analyze query patterns and identify optimization opportunities.
Database schema successfully loaded. You can proceed with query analysis.
Query Analysis Options
Database Name Identifier
DEnode by: Alex Osterneck, CLA, MSCS // ai70000 Ltd. [AI-automated optimization for normalization / denormalization databases] ( a U-of-S
A name to identify this database project in DEnode.
Use sample query log file
Use the provided sample e-commerce query log file for demonstration purposes.
Or upload your own SQL Query Log File
Choose File
No file chosen
Upload a query log file from your database server. Supported formats: PostgreSQL logs, MySQL slow query logs, or general SQL query lists.
No file selected
Analyze Queries
DEnode by: ai70000 Ltd. - Database Optimization Tool


Generated 8 recommendations for 'DEnode by: Alex Osterneck, CLA, MSCS // ai70000 Ltd. [AI-automated optimization for normalization / denormalization databases] ( a U-of-S buan576 proof-of-concept) '
Optimization Recommendations
8 optimization recommendations have been generated for DEnode by: Alex Osterneck, CLA, MSCS // ai70000 Ltd. [AI-automated optimization for normalization / denormalization databases] ( a U-of-S buan576 proof-of-concept) .
INDEX
Table: order_items
Generate SQL
INDEX
Table: order_items
Generate SQL
INDEX
Table: orders
Generate SQL
INDEX
Table: orders
Generate SQL
NORMALIZE
Table: customers
Generate SQL
INDEX
Table: customers
Generate SQL
INDEX
Table: products
Generate SQL
DENORMALIZE
Table: orders
Generate SQL
Back to Query Analysis
Regenerate Recommendations
DEnode by: ai70000 Ltd. - Database Optimization Tool

DEnode by: ai70000 Ltd.

Schema Queries

Performance Benchmark

DEnode by: Alex Osterneck, CLA, MSCS / ai70000 Ltd. [AI-automated optimization for normalization / denormalization databases] (a U-of-S buan575 proof-of-concept)

## </> SQL Implementation Plan: DENORMALIZE for orders

← Back to Recommendations

### Explanation

This plan creates: 1. A view (orders_denormalized_view) that joins orders with customers 2. A materialized table option (orders_denormalized) for better query performance 3. Recommended indexes on the materialized table The view preserves data integrity while the materialized table offers better read performance.

### Caution

Denormalization may increase storage requirements and make updates more complex.

### SQL Implementation

View orders_denormalized_view

Copy SQL

```
CREATE OR REPLACE VIEW orders_denormalized_view AS
SELECT
    orders.order_id,
    orders.customer_id,
    orders.order_date,
    orders.order_status,
    orders.shipping_address,
    orders.shipping_city,
    orders.shipping_state,
    orders.shipping_zipcode,
    orders.shipping_country,
    orders.payment_method,
    orders.order_total,
    customers.customer_name AS customers_customer_name,
    customers.customer_email AS customers_customer_email,
    customers.customer_phone AS customers_customer_phone,
    customers.customer_address AS customers_customer_address,
    customers.customer_city AS customers_customer_city,
    customers.customer_state AS customers_customer_state,
    customers.customer_zipcode AS customers_customer_zipcode,
    customers.customer_country AS customers_customer_country,
    customers.date_registered AS customers_date_registered,
    customers.last_login AS customers_last_login
FROM
    orders
    LEFT JOIN customers ON orders.customer_id = customers.customer_id;
```

Materialized orders_denormalized

Copy SQL

```
CREATE TABLE orders_denormalized AS
SELECT * FROM orders_denormalized_view;

-- Create indexes on frequently queried columns
CREATE INDEX idx_orders_denormalized_id ON orders_denormalized (id);
```

**Implementation Tip:** Always test these SQL statements in a development or staging environment before applying them to production.

← Back to Recommendations

Download SQL #1 Download SQL #2

DEnode by: ai70000 Ltd.

Home Extract Schema Analyze Queries Recommendations

Performance Benchmark

DEnode by: Alex Osterneck, CLA, MSCS / ai70000 Ltd. [AI-automated optimization for normalization / denormalization databases] (a U-of-S buan575 proof-of-concept)

## </> SQL Implementation Plan: NORMALIZE for customers

← Back to Recommendations

### Explanation

The normalization plan: 1. Creates 1 new tables to extract related columns 2. Adds foreign keys to the original table 3. Migrates data to maintain relationships 4. Removes redundant columns from the original table This plan is based on column naming patterns suggesting related data.

### Caution

Normalization requires data migration and application changes.

### SQL Implementation

Create_table customers_customer

Copy SQL

```
CREATE TABLE customers_customer (
    customers_customer_id SERIAL PRIMARY KEY,
    customer_id INTEGER,
    customer_name TEXT,
    customer_email TEXT,
    customer_phone TEXT,
    customer_address TEXT,
    customer_city TEXT,
    customer_state TEXT,
    customer_zipcode TEXT,
    customer_country TEXT
);
```

Alter_table customers

Copy SQL

```
ALTER TABLE customers
ADD COLUMN customers_customer_id INTEGER,
ADD CONSTRAINT fk_customers_customers_customer
FOREIGN KEY (customers_customer_id) REFERENCES customers_customer(customers_customer_id);
```

Data_migration

Copy SQL

```
-- Step 1: Insert distinct combinations into new table
INSERT INTO customers_customer (customer_id, customer_name, customer_email, customer_phone, customer_address, customer_city, customer_state, customer_zipcode,
SELECT DISTINCT customer_id, customer_name, customer_email, customer_phone, customer_address, customer_city, customer_state, customer_zipcode, customer_country

-- Step 2: Update foreign keys in original table
UPDATE customers t
SET customers_customer_id = nt.customers_customer_id
FROM customers_customer nt
WHERE t.customer_id = nt.customer_id AND t.customer_name = nt.customer_name AND t.customer_email = nt.customer_email AND t.customer_phone = nt.customer_phone A

-- Step 3: Remove redundant columns from original table
ALTER TABLE customers
DROP COLUMN customer_id, DROP COLUMN customer_name, DROP COLUMN customer_email, DROP COLUMN customer_phone, DROP COLUMN customer_address, DROP COLUMN customer_
```

**Implementation Tip:** Always test these SQL statements in a development or staging environment before applying them to production.

## V. COLAB SOURCE-CODE AVAILABLE UPON REQUEST to: software@ai70000.pro

##END 041525  AO // ai70000, Ltd.##