From Chaos to Clarity: A Unified Data Warehouse for Legacy Health Operations

by Shai Karmani

Contents

Project Overview

This Data Warehouse was designed for a health care organization that provides in-home caregiver services to tens of thousands of clients. The organization operates in a complex environment where each business unit is responsible for recruiting caregivers, managing schedules, tracking absences, handling salaries, and issuing invoices to external payers. Every part of the operational lifecycle is handled through a legacy system built in Delphi, using Paradox as its underlying database engine.

The legacy setup presented multiple challenges. Each branch or business unit maintained its own isolated Paradox database with no shared synchronization or integration. There was no single source of truth, and the organization had no way to view data across units in a unified way.

This project addressed that need by creating a full-scale Data Warehouse that extracts, cleans, transforms, and unifies all data sources into one consistent model. The warehouse consolidates operational data from over forty different Paradox databases and organizes it into structured layers that support analytics and reporting tools, including Power BI and other downstream systems.

For the first time, the organization now has access to a complete and reliable view of all operations across all business units.

Challenges and Solutions

Building a centralized Data Warehouse from over forty isolated legacy systems was full of challenges. Below are some of the major issues I encountered and the solutions I implemented to keep the pipeline stable and fully automated:

1. Corrupted Paradox files
The source system was based on Paradox, a file-based database prone to frequent corruption. To handle this, I identified a legacy Paradox utility that could be triggered via the command line. Before each data extraction, I executed a step that scanned and repaired all local tables, rebuilt broken indexes, and ensured the database was in a stable state before continuing the flow.

2. Incompatibility with standard SSIS execution
SSIS packages could not run normally against Paradox. To overcome this, I used the DTEXEC tool from an older version of SQL Server to trigger the packages from command line. This required avoiding standard features like Derived Column or parameterized metadata. Instead, I injected dynamic SQL expressions directly into the Data Flow components and handled transformations inside the queries themselves.

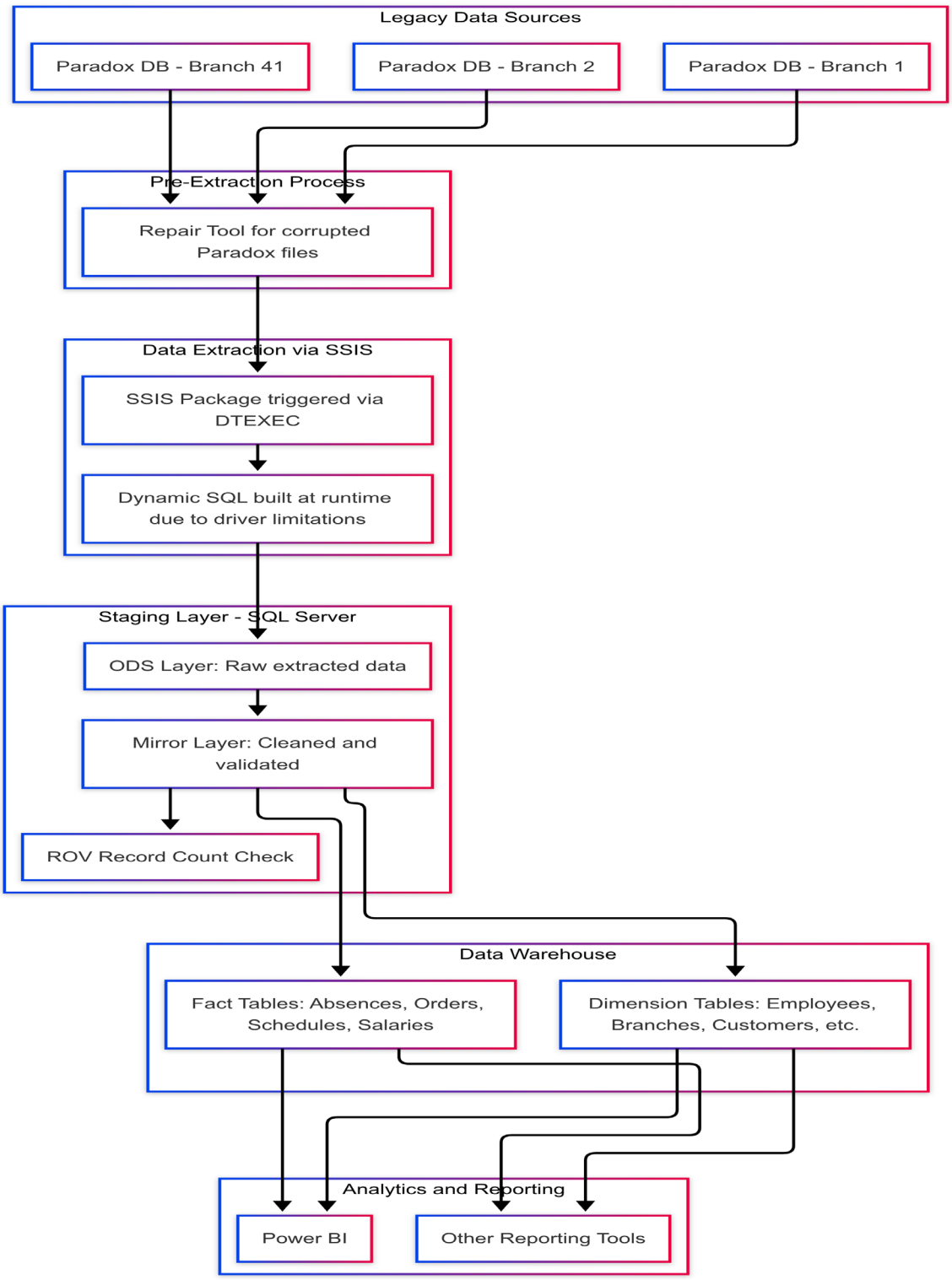
3. Distributed data across unmanaged folders
Since each business unit maintained its own Paradox database in a separate folder on a different machine, I created a control table in SQL Server that listed all **database** locations, connection settings, and status flags. The extraction process used this table to dynamically loop through all available sources and connect to them using flexible connection strings.

4. Lack of a unified schema
The legacy data structure was inconsistent between branches. I added validation steps during the staging phase that aligned and normalized each table before moving the data into the warehouse. When necessary, I included custom mapping rules to adjust for missing or renamed columns.

5. Character encoding issues
The original system used non-standard character encodings that produced unreadable text. I added a repair function that cleaned up corrupted strings and ensured all output used a consistent and readable encoding for modern reporting tools.

High-Level Architecture of the Data Warehouse Pipeline

This diagram shows the full lifecycle of data as it flows from over forty decentralized Paradox databases into a unified Data Warehouse. The process includes file-level repair, dynamic SSIS extractions, multi-layered staging, and structured dimensional and fact models. At the end of the pipeline, data is consumed by Power BI and other tools for reporting and business analysis.



Step 1: Branch-Level Data Extraction from Paradox (file system database)

The first step of the pipeline is responsible for extracting data from **41 separate Paradox databases**.
Each database represents a single branch in the organization and is stored locally as a legacy file-based structure. These files originate from an operational system built in Delphi.

The process begins by querying a configuration table in SQL Server that defines which branches should be processed. For each selected branch, the system reads the associated file path and extraction status.

A For Each Loop container iterates through all valid entries. During each iteration:

- The local path to the Paradox database is dynamically assigned to the connection string.
- The SQL query used for data extraction is constructed dynamically using expressions. This is required due to the limitations of the ODBC driver and the inability to rely on predefined metadata structures.
- Data is extracted from all relevant tables inside the branch database.

After extraction, the raw data undergoes multiple integrity checks and encoding repair procedures. The Paradox format often returns corrupted or improperly encoded characters. A custom logic block is used to fix these issues during runtime and ensure consistent data representation.

Due to the limitations of the source format, standard SSIS transformations such as Derived Column were not usable. Instead, all required logic was implemented using custom expressions and conditional components.

Once the data is validated and corrected, it is loaded into a central staging area in SQL Server, known as the ODS layer. This staging area preserves the raw structure of the data and serves as the foundation for all further transformations in the pipeline.

Step 2: Transformation and Load from ODS to Mirror Layer

The second step in the pipeline is responsible for transferring data from the operational staging layer (ODS) into the mirror layer. This step not only moves data between layers, but also includes transformation logic and critical data integrity validation.

The process begins by selecting all relevant entities from the ODS. For each entity, the data is extracted and transformed into a normalized structure that conforms with the business rules of the mirror layer. This transformation may include column renaming, type conversions, and enrichment using lookup tables.

Once the transformation is complete, the data is inserted into the mirror tables. At the end of each load process, a control query is executed to update a row count reference table. This reference acts as a validation point and is used later in the pipeline to verify data completeness. The system compares the number of rows extracted from the ODS against the number of rows loaded into the mirror. If any discrepancy is found, the pipeline raises a data quality alert.

This step also includes fallback logic to prevent duplicate insertions or data loss in case of partial failures. It is fully auditable and logs every action in a control table for monitoring and diagnostics.

The mirror layer serves as the structured and cleaned version of the data, and it is the basis for further enrichment and aggregation in the warehouse layer.

Step 3: Building and Enriching the Dimensions

The third step in the pipeline is focused on building the dimension tables in the Data Warehouse layer. These dimension tables provide the descriptive context for the facts that will be loaded later. Each dimension is extracted, transformed, and loaded through a dedicated SSIS package that ensures the data is properly shaped, cleaned, and enriched before reaching its final destination.

The dimensions included in this layer are:

DIM_Branches

This dimension contains structural and geographic details about each organizational branch. It is used to link operational records to a physical location or administrative unit. The data is extracted from the mirror layer and loaded into the warehouse after deduplication and normalization.

DIM_Committee

This table represents the various committees that authorize or monitor certain client activities. The SSIS package responsible for this dimension ensures consistent naming and handles any missing data through default value injection or referential alignment.

DIM_Customers

The customer dimension is central to the warehouse and holds demographic and status information for each individual in the system. This package includes transformation logic to unify duplicated identities and group related customer records under consistent business keys.

DIM_Employees

Employee data is enriched to include roles, status, and sometimes department codes. The logic in this package ensures that inactive or temporary employees are handled appropriately. It also standardizes dates and formats to ensure usability in analytic models.

DIM_PayingGroup

This dimension links customers to the financial entities responsible for covering their expenses. It is essential for cost tracking and funding analysis. The package contains logic to resolve mismatches between customer IDs and their paying party and ensures every customer is assigned to exactly one group.

DIM_Smalim (Pay Symbols)

This dimension contains all symbolic representations related to salary structures. It includes transformation logic to decode and classify pay codes, aligning them with internal payroll standards. This table is especially useful in dashboards or reports involving employee compensation metrics.

Step 4: Fact Table Construction and Loading

The fourth step of the pipeline focuses on building the fact tables that hold the core transactional and operational data of the system. These tables serve as the analytical backbone for all business reporting, and each fact table is created using a dedicated SSIS package.

Each package extracts pre-validated data from the mirror layer, applies final transformations, and loads the results into the corresponding fact table in the warehouse layer. This approach ensures clean, consistent, and performant data that supports downstream Power BI reports and decision-making tools.

Below is a short description of each fact table:

Fact_BL

This table stores financial charges issued to external parties who are responsible for paying on behalf of customers. The SSIS package performs aggregations and business rule filtering before loading the amounts, ensuring that each billing record is correctly attributed to the relevant payer group.

Fact_CustomersAbsence

This table captures customer absence events. The pipeline identifies periods in which customers were not available to receive services and categorizes the absence reasons. These records help track service gaps and compliance with service agreements.

Fact_EmployeesDailyLogs

This table records daily logs for field employees. Each entry represents a single workday and includes key metrics such as time on site, location, and status. The package ensures that overlapping records are resolved and that time data is normalized for analysis.

Fact_EmployeesAbsence

This table contains employee absence records. The SSIS package validates the absence periods, aligns them with HR calendars, and flags overlapping entries. These records feed directly into availability reports and workforce planning dashboards.

Fact_Goals

This table holds performance targets assigned to each operational unit. The SSIS package loads unit-specific goals such as service hours, visit counts, or coverage percentages. This data is later used for benchmarking and performance evaluation.

Fact_LastSalaryUpdate

This fact table captures the most recent salary information for employees. It stores the latest approved salary records per employee and serves as a reference point for compensation reports.

Fact_Orders

This table represents the actual service orders assigned to customers. Each row details a single service engagement, including the type of service, assigned caregiver, and approval status. The package applies business logic to deduplicate and enrich the records.

Fact_Schedule

This table contains the full scheduling information for services provided. It aligns customer orders with caregiver assignments and validates time ranges to avoid conflicts. This table is essential for analyzing service coverage, caregiver workloads, and operational efficiency.

Fact_ToSalary

This table tracks all service entries that were processed and forwarded to the payroll system. Each record confirms that a service interaction was successfully logged and paid for. The package includes validations to ensure only eligible entries are passed forward.

Final Thoughts

While this pipeline was built around a legacy system, working with Paradox databases, old Delphi-based applications, and decentralized branch-level data was not just a technical challenge , it was a mindset challenge.
The limitations pushed me to think creatively, build resilient processes, and solve problems that modern cloud platforms often abstract away.

Today, with tools like Azure Data Factory, DBT, Snowflake, and even native SQL Server integration services in the cloud, building pipelines can be faster, cleaner, and more scalable.

But what this project gave me is something deeper , the ability to work under constraints, design systems that are reliable even when nothing is standard, and treat data engineering as a craft.

These lessons transfer directly into the modern data stack.

When you've built strong architecture around chaos, you bring that clarity and structure into every cloud-based solution you build afterward.