# Expert Dossier: Group 1

# Topic: Modern Data Architecture & Data Serving

## The Paradigm Shift: From ETL to ELT in the Cloud

### The Historical Context: Resource Scarcity

For decades, the design of data integration pipelines was dictated by the physical limitations of on-premise hardware. In traditional data centers, computing power and high-speed storage were scarce and expensive commodities. This environment necessitated the **ETL (Extract, Transform, Load)** pattern. In an ETL workflow, data is extracted from source systems and processed on a dedicated, intermediate server—often referred to as the "ETL Engine"—before being written to the Data Warehouse.

The primary goal of this architecture was to curate data *before* it consumed expensive storage resources. Data Engineers had to define strict schemas (Schema-on-Write) and aggregate data to reduce volume. For example, rather than storing every single retail transaction, an ETL job might only load the "Daily Sales Total" into the warehouse. While efficient for storage, this approach resulted in a loss of granular detail and high rigidity; if a business analyst later needed the individual transaction data to analyze basket composition, the engineering team would have to rebuild the entire pipeline to extract the omitted data.

### The Cloud Era: Separation of Compute and Storage

The transition to cloud architectures (e.g., Snowflake, Google BigQuery, Amazon Redshift, Databricks) introduced a fundamental architectural change: the **separation of Compute and Storage**.

- **Storage:** Data is stored in Object Storage (e.g., AWS S3, Azure Blob Storage), which is incredibly cheap, durable, and infinitely scalable.
- **Compute:** Query processing power is provided by transient clusters (e.g., Snowflake Virtual Warehouses) that can spin up, scale out to process massive workloads, and spin down to zero when idle.

This economic shift invalidated the constraints that necessitated ETL, giving rise to **ELT (Extract, Load, Transform)**. In an ELT pipeline, the "Load" phase precedes the "Transform" phase. Data is extracted from sources and "dumped" immediately into the Data Warehouse in its rawest form—often as JSON, Avro, or Parquet files—into a "Staging" or "Raw" layer.

### Technical Advantage: Schema-on-Read

The most significant engineering benefit of ELT is agility through **Schema-on-Read**. Since the raw data is persisted in the warehouse, transformations are not destructive. If a source system changes its data structure (Schema Drift)—for example, an IoT sensor adds a new `temperature_celsius` field to its JSON log—the extraction process does not break. The data is simply loaded as a variant/JSON column. A Data Engineer can then update the SQL transformation logic to parse this new field downstream, without needing to re-extract historical data.

## The Convergence Architecture: The Data Lakehouse

### The Two-Tier Problem

Until recently, organizations faced a binary choice for data storage, leading to a complex two-tier architecture:

1. **Data Warehouses:** Databases optimized for structured data and SQL. They offer **ACID** (Atomicity, Consistency, Isolation, Durability) compliance, ensuring data integrity during concurrent updates. However, they historically struggled with semi-structured data (like video, audio, or complex logs) and were expensive for massive volumes.
2. **Data Lakes:** Repositories built on open file formats (like Apache Parquet or ORC). They excelled at storing massive volumes of raw data for Data Science and Machine Learning. However, they lacked transactional guarantees. If a pipeline crashed while writing a file, the Data Lake might be left with corrupt or partial data ("The Data Swamp"). Updating a single record in a Data Lake was architecturally impossible without rewriting entire files.

### The Solution: The Data Lakehouse

The **Data Lakehouse** is a modern architectural pattern that implements Data Warehouse features (ACID transactions, indexing, schema enforcement) directly on top of low-cost Data Lake file storage. This is achieved through **Open Table Formats** such as **Delta Lake**, **Apache Iceberg**, or **Apache Hudi**.

### How it Works: The Transaction Log

Standard Data Lakes consist of immutable files (e.g., Parquet files in a folder). You cannot modify a Parquet file; you can only replace it. Lakehouse technologies introduce a **Metadata Layer**—essentially a transaction log (e.g., the `_delta_log` folder in Delta Lake)—that sits alongside the data files.

- **ACID Transactions:** When a user executes an `UPDATE` or `DELETE` command via SQL, the engine does not modify the existing file. Instead, it writes a *new* file containing the changed data and records this action in the transaction log (a technique known as **Copy-on-Write** or **Merge-on-Read**). The engine then knows to ignore the old file and read the new one for subsequent queries.
- **Time Travel:** Because the old files are not immediately deleted but simply "dereferenced" by the log, users can query the data state as it existed at a specific timestamp. For example: `SELECT * FROM factory_logs TIMESTAMP AS OF '2025-11-20 08:00:00'`. This is invaluable for auditing and debugging pipelines.

## The "Last Mile": Modern Data Serving

### The "Headless" Semantic Layer (Metrics Store)

A chronic failure mode in Business Intelligence is **Metric Divergence**. This occurs when logic is embedded in the visualization tool rather than the data platform.

- *Example:* The Marketing team builds a dashboard in Tableau and defines `Gross Margin` as `Sales - COGS`. The Finance team builds a report in Power BI but defines `Gross Margin` as `Sales - COGS - Returns`. Even if they use the same database, they report different numbers to the CEO.

The architectural solution is a **Semantic Layer** (or Metrics Store). This is a middleware component that sits *between* the Data Warehouse and the downstream consumers (BI tools, Python scripts, AI models).

- **Abstraction:** Data Engineers define metrics and dimensions in code (typically YAML or SQL) within the Semantic Layer.
- **Query Generation:** When a user drags "Gross Margin" onto a Tableau canvas, Tableau does not generate the SQL. It asks the Semantic Layer for the data. The Semantic Layer compiles the correct SQL based on the centralized definition and sends it to the Data Warehouse. This ensures mathematical consistency across every tool in the enterprise.

**Reverse ETL: Operationalizing Data**

Dashboards are passive; they rely on humans to interpret data and take action. Modern data serving extends to **Reverse ETL**, which is the process of syncing enriched data from the Data Warehouse back into operational systems of record (SaaS).

- **Use Case:** Consider a "Customer Health Score" calculated in the Data Warehouse using complex logic (combining support tickets, usage logs, and payment history). A Support Agent working in **Zendesk** needs this score *before* picking up the phone. Reverse ETL pipelines check the Warehouse for changes to the score and push updates directly into the Zendesk database fields via API. This turns the Data Warehouse from a retrospective reporting archive into a proactive operational engine.

# Check Task: The Wagner Case Study (Group 1)

*Analyze the following scenario. You must agree on a detailed solution to present to the other groups in Round 3.*

**The Scenario:** Wagner Technologies is building its next-generation data platform on **Snowflake** (Cloud DWH) and **AWS S3** (Object Storage).

1. **The IoT Challenge:** Their new factory machines generate **500 GB of sensor logs per day**. The data arrives as nested **JSON files**. The engineering team frequently updates the firmware, which adds or renames fields in the JSON (Schema Drift).
2. **The Analytics Conflict:**
    - **Data Scientists** want access to the raw sensor data to build predictive maintenance models (predicting when a machine will break).
    - **The CFO** needs a precise "Monthly Machine Uptime Report" for accounting. She requires strict data accuracy and the ability to correct (update) erroneous logs if a sensor malfunctions.
3. **The Trust Gap:** The Sales Director ignores the official BI reports because "The revenue numbers in the dashboard never match what I see in Salesforce."

**Your Task:**

1. **Architecture Selection (Lakehouse):** Explain why a pure Data Warehouse approach (loading JSON into structured tables) would fail for the IoT data given the "Schema Drift." Propose a **Lakehouse architecture**. How does using a format like **Delta Lake** or **Iceberg** allow you to satisfy *both* the Data Scientist (raw access) and the CFO (updates/corrections) on the same data?
2. **ELT Justification:** Why is **ELT** strictly superior to ETL for the 500GB daily volume? Discuss the implications of "Schema-on-Read" for handling the firmware updates without crashing the pipeline.

3. **Solving Metric Divergence:** Propose a **Semantic Layer** solution for the Revenue problem. Where should the definition of "Revenue" live? How does this prevent the Sales Director's frustration?
4. **Reverse ETL Application:** Design a workflow to help the Sales team. If the Data Science model predicts a machine failure for "Customer X," how do we get that alert into **Salesforce** so the Account Manager can call the customer proactively?

# Further Reading & Resources

- **On the Data Lakehouse:**
  - Databricks Blog: *What is a Lakehouse?* (https://www.databricks.com/blog/2020/01/30/what-is-a-data-lakehouse.html)
  - Snowflake Architecture: *Key Concepts & Architecture* (https://docs.snowflake.com/en/user-guide/intro-key-concepts)
- **On The Semantic Layer:**
  - dbt Labs: *The Metric Layer* (https://www.getdbt.com/product/semantic-layer)
  - Benn Stancil: *The Missing Piece of the Modern Data Stack* (https://benn.substack.com/p/metrics-layer)
- **On Reverse ETL:**
  - Hightouch: *What is Reverse ETL?* (https://hightouch.com/blog/reverse-etl)

# Expert Dossier: Group 2

# Topic: Extraction Strategies & Change Data Capture (CDC)

## The Extraction Paradox: Freshness vs. Stability

The first stage of the data pipeline, **Extraction**, involves pulling data from operational Source Systems (OLTP) such as ERP databases, CRM APIs, or production logs.This process introduces a fundamental tension: Data Engineers want high-frequency extraction to provide near real-time analytics ("Data Freshness"), while Database Administrators (DBAs) want to minimize the load on the operational system to ensure it remains responsive for day-to-day business ("System Stability").

### The Full Load Strategy (Snapshot Extraction)

In a Full Load strategy, the entire dataset is extracted from the source table during every pipeline run. The target table in the Data Warehouse is usually truncated (wiped) and overwritten.

- **Mechanism:** `SELECT * FROM Source_Table;`
- **The Scalability Wall:** While simple to implement, this approach fails as data volume grows. Extracting a 50 GB table to capture 50 MB of changes wastes massive amounts of I/O bandwidth and network resources. Worse, executing a massive `SELECT` query often places a **Read Lock** on the source database table, potentially preventing the writing of new orders while the extraction is running.

### The Incremental Load Strategy (Delta Extraction)

To resolve these issues, modern pipelines utilize Incremental Loading. This involves extracting *only* the records that have changed (Inserted, Updated, or Deleted) since the last successful extraction. To implement this, the system must accurately track the state of data changes. This set of techniques is known as **Change Data Capture (CDC)**.

# Change Data Capture (CDC) Methodologies

### Method A: Query-Based CDC (Watermarking)

This is the most common "entry-level" CDC method. It relies on the existence of audit columns in the source schema, typically named `UPDATED_AT` or `MODIFIED_TIMESTAMP`.

- **Mechanism:** The data pipeline tracks a "High Water Mark"—the maximum timestamp processed in the previous run. The extraction query requests only rows where the timestamp is greater than this mark: `SELECT * FROM Orders WHERE UPDATED_AT > '2025-11-28 14:00:00';`
- **The "Hard Delete" Failure Mode:** This method has a critical flaw. If a record is physically deleted from the source ("Hard Delete"), it vanishes without updating any timestamp. The extraction query will simply miss it. Consequently, the Data Warehouse will retain the record forever, leading to "Ghost Data" (e.g., reporting revenue for orders that were actually cancelled and deleted).

### Method B: Log-Based CDC (The Gold Standard)

To solve the limitations of query-based extraction, enterprise pipelines use **Log-Based CDC**. This method does not query the tables directly. Instead, it reads the database's **Write-Ahead Log (WAL)** (in PostgreSQL), **Redo Log** (in Oracle), or **Binary Log** (in MySQL).

- **How Databases Work:** Before a database writes a change to the physical disk (the `.mdf` or data file), it *first* writes the transaction to a sequential log file. This ensures that if the power fails, the database can "replay" the log to recover the data.
- **The CDC Mechanism:** Tools like **Debezium**, **Fivetran**, or **Qlik Replicate** act as a "follower" to the database. They tap into this log stream and convert the binary log entries into structured events (e.g., `{ "op": "update", "before": {...}, "after": {...} }`).
- **Advantages:**
    1. **Completeness:** The log captures *every* event, including Hard Deletes.
    2. **Low Impact:** Reading a sequential log file is extremely lightweight compared to running complex SQL queries on table indexes.
    3. **Real-Time:** Changes can be streamed immediately as they occur, rather than waiting for a batch window.

# Extraction from Application APIs (SaaS)

Data stored in SaaS platforms (e.g., Salesforce, HubSpot, Zendesk) is not accessible via database connections. It must be extracted via **REST APIs**. This introduces specific engineering challenges associated with distributed systems over the public internet.

**Challenge 1: Pagination**

APIs rarely return large datasets in a single response to prevent server overload. Instead, they "paginate" the data.

- **The Algorithm:** The extraction script must request page 1, parse the "Next Page Token" (cursor) from the response header, and loop recursively until no token remains. If the loop logic is flawed, the pipeline might silently drop the last 10% of the data.

**Challenge 2: Rate Limiting and Backoff**

APIs enforce strict limits (e.g., "Max 500 requests per minute"). If an extraction script exceeds this, the API returns an **HTTP 429 (Too Many Requests)** error.

- **Naive Approach:** The script crashes upon receiving a 429 error.
- **Robust Approach (Exponential Backoff):** The script catches the 429 error and pauses execution. It waits for a calculated duration that increases exponentially with each failure (e.g., 1s, 2s, 4s, 8s) before retrying. This "Jitter" allows the target system to recover without overwhelming it with retry storms.

# Check Task: The Wagner Case Study (Group 2)

*Analyze the following scenario. You must agree on a detailed solution to present to the other groups in Round 3.*

**The Scenario:** Wagner Technologies is struggling with its data ingestion.

1. **The ERP Bottleneck:** The core "Orders" table (50 million rows) resides in a legacy **Oracle Database**. The current extraction runs a `SELECT *` every night at 02:00 AM. As transaction volume grows, this query is taking 5 hours, causing locks that prevent the warehouse shift from processing morning shipments.
2. **The "Ghost Order" Issue:** Occasionally, customer service deletes erroneous test orders from the ERP. The BI dashboard still shows these orders as valid revenue because the extraction pipeline never receives a signal that the rows were deleted.
3. **The CRM Crash:** A Python script extracts customer data from the Salesforce API. Every Monday morning, when volume is high, the script crashes with `Error: 429 Client Error: Too Many Requests`.

**Your Mission:**

1. **Critique the Strategy:** Explain to the IT Manager why the "Full Load" strategy on the Oracle ERP is technically unsustainable. What specific database resource (I/O, locks, CPU) is being exhausted?
2. **CDC Solution:** You cannot add triggers to the Oracle DB (performance concerns). Design a **Log-Based CDC** architecture. Which specific Oracle component will you read? How does this solve the "Ghost Order" (deletion) problem?
3. **API Engineering:** Design a pseudo-code logic for the Salesforce extraction that implements **Exponential Backoff**. How should the script react when it encounters an HTTP 429 code?

# Further Reading & Resources

- **Debezium Documentation:** *What is Change Data Capture?* (https://debezium.io/documentation/reference/stable/architecture.html)
- **Fivetran Engineering Blog:** *History of Database Replication* (https://www.fivetran.com/de/blog/history-mode-for-databases)
- **Stripe Engineering:** *Handling Rate Limits & Idempotency* (https://stripe.com/blog/rate-limiters)
- **AWS Prescriptive Guidance:** *Database Migration Strategies* (https://docs.aws.amazon.com/prescriptive-guidance/latest/strategy-database-migration/cdc.html)

# Expert Dossier: Group 3

## Topic: Transformation Logic & Data Quality Engineering

### The "T" in Modern Pipelines: Value Engineering

In the Data Engineering Lifecycle, **Transformation** is the decisive phase where technical data is elevated into business value. While Extraction handles the logistics of moving data, Transformation handles the *semantics, structure, and integrity*. In modern Cloud Data Warehouses (Snowflake, BigQuery), this is typically executed via SQL-based **ELT** (Extract-Load-Transform) pipelines, often managed by frameworks like **dbt (data build tool)**.

We can categorize transformations into three layers of increasing complexity: **Hygiene**, **Integration**, and **Business Logic**.

### Layer 1: Data Hygiene & Structural Transformation

Before data can be analyzed, it must be chemically pure and structurally sound.

#### A. Advanced Deduplication Patterns

Duplicate records are the most frequent cause of reporting errors. They arise from "At-Least-Once" delivery semantics in streaming systems (e.g., Kafka) or re-run batch jobs.

- **Naive Approach:** `SELECT DISTINCT *` is computationally expensive and insufficient if even one timestamp differs by a millisecond.
- **Robust Pattern (Window Functions):** The standard engineering pattern is to partition data by a "Business Key" (e.g., Order ID) and rank rows by "Processing Time" to keep only the latest version.

```
-- "Deduplication by Recency"
QUALIFY ROW_NUMBER() OVER (
    PARTITION BY business_key
    ORDER BY ingestion_timestamp DESC) = 1
```

### B. Structural Reshaping (Pivoting & Unpivoting)

Source systems often store data in formats hostile to SQL aggregation.

- **Unpivoting (Wide to Long):** A survey tool might store data with columns `Q1_Answer`, `Q2_Answer`, `Q3_Answer`. To analyze this, we must "Unpivot" these columns into rows (`Question_ID`, `Answer_Value`), allowing us to group by Question ID.
- **Flattening Hierarchies:** Handling JSON arrays (e.g., an order containing a list of items) requires "exploding" or "flattening" nested structures into a relational format so individual items can be summed.

### C. Type Enforcement & Sanitization

- **The "Silent Failure" of Strings:** A CSV file might contain `"1,000.00"` (string with comma) in a Price column. Casting this directly to `FLOAT` might fail or result in `1.0`. Robust transformations strip non-numeric characters before casting.
- **Whitespace & Casing:** `GROUP BY City` will treat `"Berlin"` and `" Berlin"` as distinct entities. Standardizing casing (`UPPER()`) and trimming whitespace (`TRIM()`) is mandatory for all categorical columns.

## Layer 2: Semantic Integration (The "Tower of Babel" Problem)

This layer solves the problem of integrating data from disparate systems that speak different "languages."

### A. Temporal Standardization (Time Zones)

A global warehouse receives logs from servers in Tokyo (JST), Berlin (CET), and New York (EST).

- **The Error:** Storing local times. An event at 10:00 AM Tokyo happened *before* 04:00 AM Berlin, but naive SQL comparison suggests the opposite.
- **The Standard:** All timestamps must be converted to **UTC (Coordinated Universal Time)** during the transformation phase. Local time conversion happens only at the very last mile (in the BI tool) for the user's convenience.

### B. Currency Conversion

Aggregating revenue across regions requires a common currency. This is not a simple multiplication; it is a **Slowly Changing Dimension** problem.

- **The Pattern:** You cannot use today's exchange rate for sales made last year. The transformation must join the Sales Fact table with a `Currency_Rates` table on *both* `Currency_Code` and `Transaction_Date` to apply the historically accurate rate.

### C. Reference Data Mapping

Merging acquired companies often leads to code collisions.

- *System A:* Status `1` = Active.
- *System B:* Status `1` = Pending.
- **The Solution:** A centralized **Mapping Table** (Lookup). The pipeline translates local codes into a global "Super-Type" (e.g., `Global_Status_Active`, `Global_Status_Pending`) before loading the final dimension.

## Layer 3: Integrity & The "Unknown Member"

Handling **NULL** values differentiates amateur pipelines from professional ones.

- **Fact Table Foreign Keys:** If a Sales Fact references a Customer that does not exist in the Dimension table (referential integrity failure), an `INNER JOIN` drops the sale.
    - **Best Practice:** Replace `NULL` keys with `-1` (or `0`). Create a default row in the Dimension Table with `ID = -1` and `Name = "Unknown/Missing"`. This preserves the fact measure (Revenue) while explicitly flagging the missing context.
- **Measures:** `NULL` in a numeric column (e.g., `Discount_Amount`) propagates through arithmetic. `100 + NULL = NULL`. Transformations must use `COALESCE(Discount_Amount, 0)` to ensure math works safely.

## Deep Dive: Data Profiling & Quality Gates

Data Quality (DQ) is the defense layer preventing "Garbage In" from becoming "Garbage Out."

### A. The Six Dimensions of Data Quality

1. **Accuracy:** Does the data reflect reality?
2. **Completeness:** Are critical fields populated? (Null density).
3. **Consistency:** Does data match across systems?
4. **Timeliness:** Is data available when needed?
5. **Uniqueness:** Are there duplicates?
6. **Validity:** Does data conform to syntax (e.g., email regex)?

### B. Data Profiling

Profiling is the statistical analysis of data *before* logic application.

- **Cardinality Analysis:** Identifying potential Primary Keys (100% unique).
- **Distribution Analysis:** Detecting outliers (e.g., `Age = 199`).
- **Pattern Matching:** Validating string formats (e.g., ensuring all IBANs start with two letters).

**C. Automated Quality Gates (Circuit Breakers)**

Modern pipelines (using tools like **Great Expectations** or **dbt tests**) implement automated tests that run *during* the pipeline execution.

- **Warning vs. Failure:**
  - *Warning:* "Null rate for 'Phone Number' increased by 5%." (Alert the team, but let the pipeline finish).
  - *Failure (Circuit Breaker):* "Primary Key is not unique." (STOP the pipeline immediately. Do not load bad data into the warehouse).
- **Quarantine (Dead Letter Queues):** Instead of crashing on bad rows, robust pipelines divert rows failing validation logic into a "Quarantine Table" for manual inspection, while allowing the clean rows to proceed.

## Check Task: The Wagner Case Study (Group 3)

*Analyze the following scenario. You must agree on a detailed solution to present to the other groups in Round 3.*

**The Scenario:** Wagner Technologies has acquired a French competitor and is integrating their sales data (CSV files) into the Global Snowflake Data Warehouse.

1. **The Time Zone Trap:** French sales are recorded in Paris time (CET). US sales are in EST. The CFO's "Global Daily Sales" report shows revenue spikes at odd hours because the days are misaligned.
2. **The Currency Crisis:** The French file has amounts in EUR. The US system is in USD. The current pipeline simply sums them up (`SUM(amount)`), resulting in a mathematically meaningless total.
3. **The "Ghost Product" Issue:** The French file contains sales for `Product_ID = 999`. However, `999` has not yet been added to the Wagner Master Product Dimension. Currently, these sales are disappearing from the dashboard because of an Inner Join.

**Your Mission:**

1. **Temporal & Monetary Standardization:** Design the SQL logic to fix the Time and Currency issues.
   - *Time:* How do you convert the timestamps to ensure a global "Sales Day" is accurate?
   - *Money:* Describe the join logic required to convert EUR to USD accurately. Do you use today's rate or the rate on the day of the sale? Why?
2. **Handling Late-Arriving Dimensions:** How do you solve the "Ghost Product 999" issue?
   - *Option A:* Delete the sales.
   - *Option B:* Update the Product Dimension with a placeholder.
   - *Option C:* Use a "-1 Unknown" key.
   - *Argue for the best approach for the CFO (who cares about total revenue).*
3. **Designing the Defense:** Define three specific **Quality Gates** (Tests) you would add to the French pipeline to prevent future errors.
   - *One schema test.*

- *One business logic test.*
- *One consistency test.*

## Further Reading & Resources

- **Great Expectations:** *Beginner's Guide to data quality tests* (https://pages.greatexpectations.io/beginners-guide)
- **dbt Labs:** *Testing your data with dbt* (https://docs.getdbt.com/docs/build/tests)
- **Kimball Group:** *Design Tip #57: Early Arriving Facts* (https://www.kimballgroup.com/wp-content/uploads/2012/05/DT57EarlyArriving.pdf)
- **Monte Carlo Data:** *The 6 Dimensions of Data Quality* (https://www.montecarlodata.com/blog-6-data-quality-dimensions-examples/)

# Expert Dossier: Group 4

## Topic: Loading Strategies & History Management

### The "L" in ELT: Advanced Loading Patterns

In a modern Data Warehouse, the **Load** phase is the final commitment of data to storage. It is not merely about moving files; it is about maintaining **Referential Integrity**, ensuring **Idempotency** (the ability to restart a failed job without creating duplicates), and optimizing for **Write Performance**.

Unlike traditional transactional databases (OLTP) that are optimized for single-row inserts, Data Warehouses (OLAP) are optimized for **Bulk Loading**. Inserting 1 million rows one-by-one into Snowflake or BigQuery is prohibitively slow. The standard engineering pattern is to stage files (Parquet/CSV) in Object Storage and execute a bulk `COPY INTO` command.

#### A. The Merge (Upsert) Pattern

For Dimension tables (e.g., Customers, Products), we cannot simply append data; we must handle updates to existing records. The standard SQL pattern is the **MERGE** statement (often called "Upsert" - Update/Insert).

- **Logic:** The database joins the incoming staging data with the target table on the **Natural Key** (e.g., `Customer_ID`).
  - **WHEN MATCHED:** The existing row is updated with new values.
  - **WHEN NOT MATCHED:** The new row is inserted.
- **Engineering Challenge:** In distributed systems, MERGE operations are expensive because they require scanning the target table to find matches. Optimizing the "Pruning" (using partition keys) is critical to prevent full table scans.

#### B. The Immutable Append (Fact Tables)

Fact tables (Sales, Clickstreams, IoT Logs) represent events that occurred at a specific point in time. In theory, history never changes, so we use an **Append-Only** strategy.

- **The Problem:** What if late-arriving data creates duplicates?
- **The Solution:** The **Delete-Insert** pattern. Instead of a complex MERGE, pipelines often delete all records for a specific time window (e.g., `DELETE FROM sales WHERE date = '2025-11-28'`) and then insert the fresh batch for that window. This guarantees **idempotency**.

# Surrogate Key Architecture in Distributed Systems

Operational systems use **Natural Keys** (e.g., string SKU `"X-2000"`). Data Warehouses rely on **Surrogate Keys** (integers `1, 2, 3...`) to isolate analytical models from source system changes and improve join performance.

### Generating Keys in Massively Parallel Processing (MPP)

In a traditional database (PostgreSQL), a simple `SEQUENCE` object generates keys `1, 2, 3`. In a distributed Cloud DWH (like Snowflake), strictly sequential generation inhibits performance because all compute nodes must synchronize to get the "next number."

- **Approach A (Sequences):** Modern DWHs offer "gapped" sequences. Node A generates `1-100`, Node B generates `101-200`. Keys are unique but not strictly sequential.
- **Approach B (Hash Keys):** A popular modern alternative (used in Data Vault) is to generate keys deterministically using a Hash Function (e.g., `MD5(Natural_Key)`). This allows keys to be calculated in parallel without any coordination between nodes, though it results in alphanumeric strings rather than integers.

# Deep Dive: Slowly Changing Dimensions (SCD)

The most complex logic in the Loading phase is managing history. When a customer moves from Berlin to Munich, how do we reflect that?

### SCD Type 1 (Overwrite - "The Revisionist History")

We update the `City` column from "Berlin" to "Munich".

- *Result:* All historical sales made to that customer while they lived in Berlin will now appear in reports as if they were made in Munich. History is lost.
- *Use Case:* Correcting data entry errors (e.g., fixing a typo in a name).

### SCD Type 2 (Row Versioning - "The Historian")

This is the gold standard for accurate reporting. We assume every record has a lifespan.

- **Schema Requirements:** We add three technical columns:
  1. `valid_from` (Timestamp)
  2. `valid_to` (Timestamp, usually NULL or '9999-12-31' for current rows)
  3. `is_current` (Boolean Flag)
- **The Logic:** When a change is detected for Customer A:
  1. **Close the old record:** Update the existing row where `is_current = TRUE`. Set `valid_to = Current_Timestamp` and `is_current = FALSE`.

2. **Open the new record:** Insert a new row with the new data. Set `valid_from = Current_Timestamp`, `valid_to = NULL`, `is_current = TRUE`.*Note:* This requires a new Surrogate Key for the new row.

**SCD Type 4 (Mini-Dimensions - "The Performance Optimizer")**

If an attribute changes very frequently (e.g., "Customer Risk Score" changes daily), applying Type 2 logic creates massive explosion in the Customer Dimension table (millions of rows for one customer).

- **The Solution:** Split the fast-changing attributes into a separate "Mini-Dimension" (e.g., `Dim_Risk_Profile`). The Fact table now references two dimensions: the static `Dim_Customer` and the volatile `Dim_Risk_Profile`.

## Advanced Challenge: The "Late Arriving Dimension"

What happens if the ETL pipeline processes a Sales Fact record for Product X, but the Dimension pipeline hasn't created Product X yet?

- **The Failure:** The `JOIN` fails, dropping the revenue.
- **The Engineering Pattern:** "Inferred Members." The Fact loader detects the missing key and *inserts* a placeholder row into the Product Dimension (e.g., "Product X (Pending Details)"). When the Dimension pipeline finally runs, it updates this placeholder with the full descriptive details (Color, Size, etc.).

## Check Task: The Wagner Case Study (Group 4)

*Analyze the following scenario. You must agree on a detailed solution to present to the other groups in Round 3.*

**The Scenario:** Wagner Technologies sells huge industrial presses. The sales cycle is long (6-12 months).

1. **The Deal Structure:** A sales opportunity goes through stages: "Lead" -> "Negotiation" -> "Contract" -> "Closed".
2. **The Reporting Requirement:** The VP of Sales wants to see a "Funnel Analysis": How long did Deal #505 spend in the "Negotiation" stage compared to the "Contract" stage?
3. **The Data Issue:** Currently, the Data Warehouse uses **SCD Type 1** for the `Dim_Opportunity` table. It only shows the *current* stage. We have lost the timing of previous stages.

**Your Mission:**

1. **SCD Strategy:** Explain why the current Type 1 strategy makes the Funnel Analysis impossible. Propose an **SCD Type 2** implementation. Sketch the columns (`valid_from`, etc.) for Deal #505 as it moves from "Negotiation" to "Contract".
2. **Handling "Corrections":** A sales rep accidentally changes a deal to "Closed" and then immediately changes it back to "Negotiation" 5 minutes later.
    - If your pipeline runs hourly, will you see this change?
    - If your pipeline runs every minute, do you *want* to store this error as a permanent historical row? Discuss how to filter "noise" vs "true history".
3. **Surrogate Keys:** When Deal #505 moves to the new stage, a new dimension row is created with a new Surrogate Key. The **Fact Table** (Sales) tracks the daily value of the deal. Which Surrogate Key should the Fact Table link to for *yesterday's* snapshot? The old one or the new one?

## Further Reading & Resources

- **Snowflake Documentation:** *Merging Data (Upsert)* (https://docs.snowflake.com/en/sql-reference/sql/merge)
- **dbt Labs:** *Snapshotting data (SCD Type 2 in dbt)* (https://docs.getdbt.com/docs/build/snapshots)
- **Kimball Group:** *Design Tip #152: Slowly Changing Dimensions Types 0, 4, 5, 6 and 7* (https://www.kimballgroup.com/2013/02/design-tip-152-slowly-changing-dimension-types-0-4-5-6-7/)