Data Warehousing Essentials

A Comprehensive Guide to Modern Analytics Infrastructure

Master the core concepts that power enterprise data systems

2025 Edition

1. OLTP vs OLAP

Understanding the fundamental difference between transactional and analytical systems is crucial for designing effective data architectures.

OLTP: Online Transaction Processing

OLTP systems are designed to handle day-to-day business operations. They excel at processing high volumes of small, quick transactions such as recording sales, updating inventory, or processing customer orders.

> **Key Characteristics:**
>
> - **Operation Focus:** INSERT, UPDATE, DELETE operations
> - **Query Pattern:** Simple queries affecting few records
> - **Schema Design:** Highly normalized (3NF) to eliminate redundancy
> - **Response Time:** Milliseconds required for user satisfaction
> - **Users:** Thousands of concurrent users
> - **Data Volume:** Current operational data (days to months)

OLAP: Online Analytical Processing

OLAP systems are optimized for complex analytical queries that aggregate large volumes of historical data. They power business intelligence, reporting, and data analysis workloads.

> **Key Characteristics:**
>
> - **Operation Focus:** Complex SELECT queries with aggregations
> - **Query Pattern:** Queries scanning millions of records
> - **Schema Design:** Denormalized (Star/Snowflake) for query performance
> - **Response Time:** Seconds to minutes acceptable
> - **Users:** Dozens to hundreds of analysts
> - **Data Volume:** Historical data (years of records)

Aspect	OLTP	OLAP
Purpose	Run business operations	Analyze business performance
Data Updates	Frequent, real-time	Periodic batch updates

Transaction Size	Small (few records)	Large (millions of records)
Database Size	Gigabytes to Terabytes	Terabytes to Petabytes

Why Separate Systems? Organizations maintain separate OLTP and OLAP systems because their requirements conflict. Running complex analytical queries on a transactional database would slow down critical business operations. Conversely, the frequent updates of OLTP would make analytical queries unpredictable and inefficient.

2. ETL vs ELT

The method you choose for moving data from source systems to your warehouse significantly impacts performance, flexibility, and architecture.

ETL: Extract, Transform, Load

The traditional approach where data is transformed before loading into the warehouse.

Extract → Transform → Load

> **Process Flow:**
>
> 1. **Extract:** Pull data from source systems
> 2. **Transform:** Clean, validate, and transform data in a staging area or ETL server
> 3. **Load:** Insert transformed data into warehouse

When to Use ETL:

- **Legacy Systems:** On-premise warehouses with limited compute
- **Sensitive Data:** PII must be masked before loading
- **Complex Transformations:** Heavy processing required before storage
- **Network Constraints:** Limited bandwidth between systems
- **Data Quality:** Must validate before loading

ELT: Extract, Load, Transform

Modern approach leveraging the warehouse's processing power to transform data after loading.

Extract → Load → Transform

> **Process Flow:**
>
> 1. **Extract:** Pull raw data from sources
> 2. **Load:** Load raw data directly into warehouse
> 3. **Transform:** Use warehouse compute to transform data in place

When to Use ELT:

- **Cloud Warehouses:** Snowflake, BigQuery, Redshift with scalable compute

- **Big Data:** Processing massive volumes efficiently
- **Flexibility:** Need to reprocess with different transformation logic
- **Speed:** Faster initial load, transform only what's needed
- **Data Lake Integration:** Store raw data for multiple use cases

Factor	ETL	ELT
Processing Location	External ETL server	Inside warehouse
Time to Load	Slower (transform first)	Faster (raw load)
Flexibility	Less (retransform harder)	More (raw data available)
Best For	Traditional warehouses	Cloud data platforms

Modern Trend: ELT has become dominant with cloud warehouses offering virtually unlimited compute power. Tools like dbt (data build tool) have made SQL-based transformations inside warehouses the standard approach.

3. Star Schema & Snowflake Schema

Dimensional modeling provides the foundation for organizing data warehouses to optimize analytical queries.

Star Schema

The star schema is named for its visual appearance: a central fact table surrounded by dimension tables, resembling a star.

> **Structure:**
> - **Fact Table:** Center of the star containing measurements and foreign keys
> - **Dimension Tables:** Points of the star with descriptive attributes
> - **Denormalized:** Dimension tables contain redundant data
> - **Simple Joins:** One-hop from fact to any dimension

Advantages of Star Schema:

- **Query Performance:** Fewer joins mean faster queries
- **Simplicity:** Easy for analysts and BI tools to understand
- **Predictable:** Query patterns are straightforward
- **Optimized:** Database optimizers work well with this pattern

Disadvantages:

- **Storage:** Data redundancy increases storage needs
- **Updates:** Denormalized data harder to update consistently
- **Integrity:** More potential for data inconsistencies

Snowflake Schema

The snowflake schema extends the star schema by normalizing dimension tables into multiple related tables.

> **Structure:**
> - **Normalized Dimensions:** Dimension tables split into subdimensions
> - **Hierarchical:** Represents hierarchies explicitly (Country → State → City)
> - **Multiple Joins:** May require several joins to reach leaf dimensions
> - **Less Redundancy:** Reduces duplicate data storage

Advantages of Snowflake Schema:

- **Storage Efficiency:** Eliminates redundant data
- **Data Integrity:** Easier to maintain consistency
- **Hierarchies:** Explicitly models relationships

Disadvantages:

- **Complexity:** More tables and relationships to manage
- **Query Performance:** Additional joins slow down queries
- **Less Intuitive:** Harder for business users to navigate

Characteristic	Star Schema	Snowflake Schema
Normalization	Denormalized dimensions	Normalized dimensions
Join Complexity	Simple (1 level)	Complex (multiple levels)
Query Speed	Faster	Slower
Storage Space	More	Less
Maintenance	Simpler	More complex

Best Practice: Star schema is preferred in most modern cloud warehouses where storage is cheap and query performance is critical. Use snowflake schema only when storage costs are prohibitive or data integrity requirements demand it.

4. Fact & Dimension Tables

Understanding the distinction between facts and dimensions is fundamental to dimensional modeling.

Fact Tables

Fact tables store quantitative measurements of business processes. They represent business events or transactions.

> **Characteristics:**
>
> - **Measurements:** Numeric values (sales amount, quantity, duration)
> - **Foreign Keys:** References to dimension tables
> - **Granularity:** Each row represents a specific event or measurement
> - **Large Volume:** Typically the biggest tables in the warehouse
> - **Growing:** Continuously accumulates new records

Types of Fact Tables:

1. Transaction Fact Tables

Record individual business events at the lowest level of detail.

Example: Each row = one sale transaction

2. Periodic Snapshot Fact Tables

Capture the state of business at regular intervals.

Example: Daily account balances, monthly inventory levels

3. Accumulating Snapshot Fact Tables

Track the progress of a process with multiple milestones.

Example: Order fulfillment (ordered → shipped → delivered)

Types of Measures:

- **Additive:** Can be summed across all dimensions (sales revenue, quantity)

- **Semi-Additive:** Can be summed across some dimensions (account balance across customers, not time)
- **Non-Additive:** Cannot be summed (ratios, percentages, averages)

Dimension Tables

Dimension tables provide descriptive context for facts. They answer who, what, when, where, why, and how questions.

> **Characteristics:**
> - **Descriptive Attributes:** Text fields describing entities
> - **Primary Key:** Surrogate key connecting to facts
> - **Relatively Static:** Changes less frequently than facts
> - **Smaller Volume:** Much fewer rows than fact tables
> - **Wide Tables:** Many columns with descriptive data

Common Dimension Types:

- **Date/Time Dimension:** Calendar attributes (day, week, month, quarter, holiday flag)
- **Product Dimension:** Product attributes (name, category, brand, SKU)
- **Customer Dimension:** Customer information (name, segment, location, demographics)
- **Location Dimension:** Geographic hierarchy (city, state, region, country)
- **Employee Dimension:** Staff details (name, department, role, manager)

Design Principle: Facts contain measurements that change frequently. Dimensions contain descriptive context that changes slowly. This separation enables efficient storage and optimal query performance.

Surrogate Keys: Always use surrogate keys (auto-generated integers) as dimension primary keys instead of natural business keys. This provides flexibility when source systems change and improves join performance.

5. Slowly Changing Dimensions (SCDs)

Dimension data changes over time, and how you handle these changes affects historical analysis accuracy.

Why SCDs Matter

When a customer moves to a new address or a product changes categories, you must decide: preserve history, overwrite data, or track both? This decision impacts reporting accuracy and data integrity.

Type 0: Retain Original

> **Strategy:** Never change the original value. It remains fixed forever.
>
> **Use Case:** Immutable attributes like date of birth, original customer registration date
>
> **Implementation:** Simply don't update the attribute

Type 1: Overwrite

> **Strategy:** Update the value in place, losing historical information.
>
> **Use Case:** Corrections of errors, attributes where history doesn't matter
>
> **Pros:** Simple, no additional storage
>
> **Cons:** Historical data lost, cannot recreate past reports accurately

Example: Customer moves and you only care about current address.

Type 2: Add New Row (Most Common)

> **Strategy:** Create a new row with the new value, keeping old row intact.
>
> **Use Case:** Full historical tracking required
>
> **Implementation:** Add effective_date, end_date, and is_current flag
>
> **Pros:** Complete history preserved, accurate point-in-time reporting

Cons: More storage, more complex queries

Example Table Structure:

customer_key	customer_id	name	city	effective_date	end_date	is_curre
1001	C123	John Smith	Boston	2020-01-01	2023-05-31	N
1002	C123	John Smith	Seattle	2023-06-01	9999-12-31	Y

Type 3: Add New Column

Strategy: Store both current and previous value in separate columns.

Use Case: Track only one prior value, limited history

Pros: Simple queries, limited history preserved

Cons: Only stores one previous value, requires schema changes

Example: current_price and previous_price columns

Type 4: History Table

Strategy: Keep current data in main table, move historical records to separate history table.

Use Case: When current queries shouldn't be impacted by historical data volume

Pros: Current table stays small and fast

Cons: More complex queries to join current and historical data

Type 6: Hybrid (Combines Type 1 + 2 + 3)

Strategy: Add new rows (Type 2) but also maintain current value column in all historical rows (Type 1) and track previous value (Type 3).

Use Case: Need both historical accuracy and easy access to current values

Best Practice: Type 2 is the gold standard for most business scenarios. It provides complete audit trails and enables accurate historical analysis while remaining

relatively simple to implement and query.

6. Data Partitioning & Clustering

Large tables require smart organization strategies to maintain query performance at scale.

Data Partitioning

Partitioning divides large tables into smaller, more manageable segments based on column values.

> **How It Works:**
> The database physically separates data into distinct partitions. When querying, only relevant partitions are scanned, dramatically reducing I/O.

Common Partitioning Strategies:

Range Partitioning

Divide data based on value ranges, most commonly by date.

Example: Partition sales data by month or quarter

- Partition 1: January 2024
- Partition 2: February 2024
- Partition 3: March 2024

List Partitioning

Divide data based on discrete values.

Example: Partition by region (North, South, East, West)

Hash Partitioning

Use a hash function to distribute data evenly across partitions.

Example: Hash customer_id to distribute load evenly

Benefits of Partitioning:

- **Query Performance:** Scan only relevant partitions (partition pruning)

- **Maintenance:** Easier to archive or delete old data
- **Loading:** Load new data into specific partitions without affecting others
- **Parallelism:** Process multiple partitions simultaneously
- **Availability:** If one partition fails, others remain accessible

Best Practice: Choose partition keys based on how data is commonly filtered in queries. Date/time is the most common partition key because most analytical queries filter by time period.

Data Clustering

Clustering physically organizes data within partitions (or entire tables) to co-locate related records.

> **How It Works:**
> Data is sorted and stored based on clustering key values. Records with similar values are stored physically close together, reducing data scan requirements.

Benefits of Clustering:

- **Min/Max Pruning:** Skip blocks where values fall outside query range
- **Better Compression:** Similar values compress more efficiently
- **Improved Joins:** Co-located data speeds up join operations
- **Reduced I/O:** Fewer disk blocks need to be read

Choosing Cluster Keys:

- Select columns frequently used in WHERE clauses
- Choose columns with moderate to high cardinality
- Consider columns used in JOIN operations
- Limit to 3-4 columns maximum (order matters)