Supplemental: Introduction to Data Warehousing

ISM 6562 - Big Data for Business Applications

Dr. Tim Smith

2026-01-01

Table of contents

Why This Document Exists

The midterm project asks you to work with a **data warehouse** built using a **star schema**. If you haven't taken a dedicated data warehousing course, some of these concepts may be unfamiliar. This supplemental reading gives you enough background to understand what the midterm's warehouse is doing and why it's designed that way.

You do not need to memorize all of this. The goal is to understand the core ideas so that the midterm assignment makes sense.

Two Kinds of Databases

Most organizations maintain two fundamentally different types of databases, each optimized for a different purpose.

Operational Databases (OLTP)

An **operational database** (also called a transactional database or OLTP system) supports the day-to-day operations of a business. When a customer places an order, when an employee clocks in, when a product's inventory is updated — these events are recorded in operational databases.

Characteristics:

- **Purpose**: Record individual business transactions as they happen
- **Users**: Applications, front-end systems, employees entering data
- **Queries**: Simple reads and writes — insert one order, update one customer, look up one product
- **Optimized for**: Fast inserts and updates on individual rows
- **Schema**: Highly **normalized** (explained below) to avoid data duplication

In the midterm, the `sales-db` and `hr-db` are operational databases. They store customers, products, orders, employees, departments, and timesheets in normalized tables.

The sales operational database has three tables — `customers`, `products`, and `orders` — with foreign key relationships:

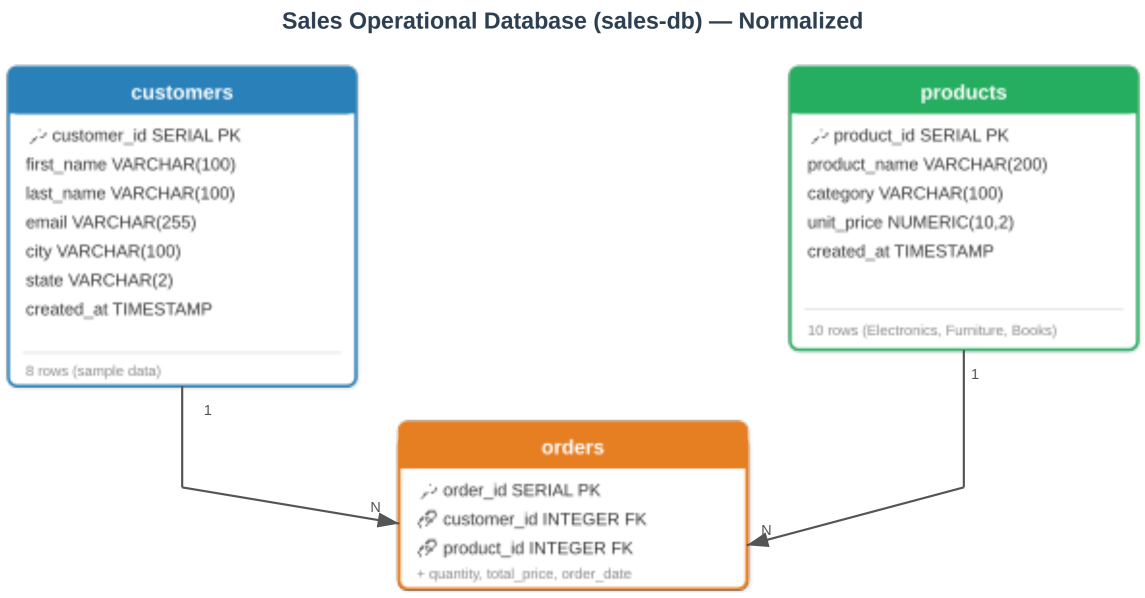


Figure 1: Sales Operational Database (sales-db) — Entity Relationship Diagram

The HR operational database has three tables — `departments`, `employees`, and `timesheets` — also with foreign key relationships:

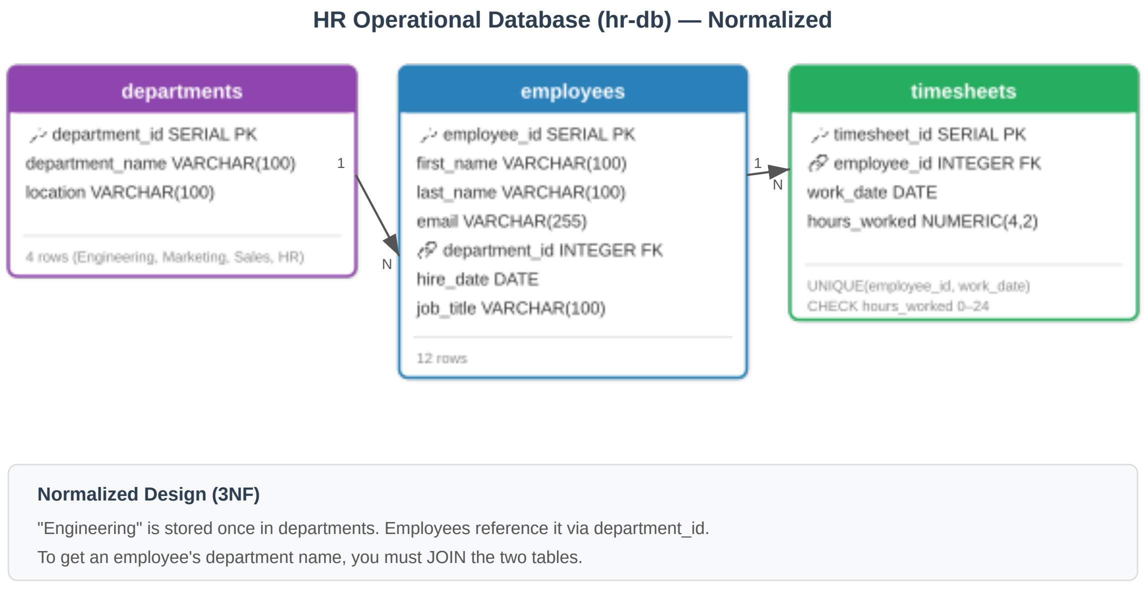


Figure 2: HR Operational Database (hr-db) — Entity Relationship Diagram

Analytical Databases (OLAP)

An **analytical database** (also called a data warehouse or OLAP system) is designed for answering business questions across large amounts of data. Rather than recording individual transactions, it's built to support queries like "What was total revenue by product category last quarter?" or "Which department has the highest employee turnover?"

Characteristics:

- **Purpose**: Support analytical queries and business intelligence
- **Users**: Analysts, managers, reporting tools, dashboards
- **Queries**: Complex reads across many rows — aggregations, joins across multiple dimensions, trend analysis
- **Optimized for**: Fast reads over large datasets (not fast writes)
- **Schema**: **Denormalized** (explained below) to minimize the number of joins needed for analytical queries

In the midterm, the `warehouse-db` is the analytical database. It stores data in a star schema designed for fast analytical queries.

Side-by-Side Comparison

Aspect	Operational (OLTP)	Analytical (OLAP)
Primary purpose	Record transactions	Analyze trends
Typical query	"Get order #1234"	"Total revenue by quarter and region"
Data freshness	Real-time	Periodically refreshed (ETL)
Schema design	Normalized (3NF)	Denormalized (star/snowflake)
Number of tables	Many (to avoid duplication)	Fewer (to reduce joins)
Query complexity	Simple (1–2 table lookups)	Complex (joins across many dimensions)
Write pattern	Many small writes	Bulk loads (ETL batches)
Read pattern	Few rows at a time	Millions of rows aggregated

Normalization vs Denormalization

Normalization (Operational Databases)

Normalization is the process of organizing data to eliminate redundancy. In a normalized database, each piece of information is stored in exactly one place. If you need to change a department's name, you change it in one row of the `departments` table, and every employee in that department automatically reflects the change.

Example from the midterm's HR operational database:

```
departments table:
  department_id | department_name | location
  1             | Engineering     | Tampa
  2             | Marketing       | Orlando

employees table:
  employee_id | first_name | last_name | department_id
  101         | Alice      | Johnson   | 1
  102         | Bob        | Smith     | 1
  103         | Carol      | Davis     | 2
```

Notice that "Engineering" and "Tampa" are stored only once (in the `departments` table). Employees reference their department via `department_id`. To get an employee's department name, you must JOIN the two tables:

```sql
SELECT e.first_name, e.last_name, d.department_name, d.location
FROM employees e
JOIN departments d ON e.department_id = d.department_id;
```

Benefits of normalization:

- No duplicated data — change a department name in one place
- Less storage waste
- Easier to maintain data consistency

Drawback: Analytical queries require many JOINs, which can be slow at scale.

Denormalization (Analytical Databases)

Denormalization is the deliberate introduction of redundancy to speed up read queries. In a denormalized table, related data is stored together so you don't need JOINs to answer common questions.

Example from the midterm's warehouse database:

```
dim_employee table:
  employee_key | first_name | last_name | department_name | department_location
  1            | Alice      | Johnson   | Engineering     | Tampa
  2            | Bob        | Smith     | Engineering     | Tampa
  3            | Carol      | Davis     | Marketing       | Orlando
```

Notice that "Engineering" and "Tampa" are repeated for both Alice and Bob. This is intentional — it means any query about employees and their departments can be answered from this single table without a JOIN:

```sql
SELECT first_name, last_name, department_name, department_location
FROM dim_employee;
```

Benefits of denormalization:

- Faster queries (fewer JOINs)
- Simpler query logic
- Better performance for aggregations across large datasets

Drawback: Data is duplicated, so updates require changing multiple rows. This is acceptable in a warehouse because data is loaded in bulk by an ETL process, not updated row-by-row.

The Star Schema

The **star schema** is the most common design pattern for data warehouses. It gets its name from its shape: a central **fact table** surrounded by **dimension tables**, like a star.

Fact Tables

A **fact table** stores the measurable events of a business — the things you want to count, sum, or average. Each row represents one business event at a specific grain (level of detail).

Key characteristics:

- Contains **measures** (numeric values): quantity, price, revenue, duration
- Contains **foreign keys** pointing to dimension tables
- Typically the largest table in the warehouse (many rows)
- Each row represents one atomic business event

In the midterm, `fact_sales` is the fact table:

```
fact_sales:
  sale_key          -- Surrogate primary key
  date_key          -- FK → dim_date
  customer_key      -- FK → dim_customer
  product_key       -- FK → dim_product
  source_order_id   -- Link back to operational database
  quantity          -- Measure: how many units
  unit_price        -- Measure: price per unit
  total_price       -- Measure: total revenue
```

Each row represents one order. The measures (`quantity`, `unit_price`, `total_price`) are the numbers you aggregate in analytical queries.

Dimension Tables

A **dimension table** stores the descriptive attributes that give context to the facts. They answer the "who, what, when, where" of each business event.

Key characteristics:

- Contains **descriptive attributes** (text, categories, dates)
- Relatively few rows compared to fact tables
- Denormalized — related attributes are stored together
- Provide the labels and groupings for analytical queries

In the midterm, there are four dimension tables:

`dim_date` — When did the sale happen?

```
dim_date:
  date_key       -- Integer key (YYYYMMDD format, e.g., 20260115)
  full_date      -- The actual date
  year, quarter, month, day_of_month
  month_name     -- "January", "February", etc.
  day_of_week, day_name, is_weekend
```

The date dimension is pre-populated with every day of the year. Pre-calculated fields like `quarter`, `month_name`, and `is_weekend` mean your queries never need date arithmetic — just filter or group by the column you need.

`dim_customer` — Who bought it?

```
dim_customer:
  customer_key         -- Warehouse surrogate key
  source_customer_id   -- Link to operational sales-db
  first_name, last_name, email, city, state
```

`dim_product` — What was bought?

```
dim_product:
  product_key         -- Warehouse surrogate key
  source_product_id   -- Link to operational sales-db
  product_name, category, unit_price
```

`dim_employee` — Who works here? (with denormalized department info)

```
dim_employee:
  employee_key           -- Warehouse surrogate key
  source_employee_id     -- Link to operational hr-db
  first_name, last_name, email, job_title
  department_name        -- Denormalized from departments table
  department_location    -- Denormalized from departments table
```

How the Star Schema Looks

The diagram below shows the complete star schema used in the midterm's warehouse. The central `fact_sales` table (red) is surrounded by four dimension tables (blue, green, orange, purple). Foreign key relationships are shown as arrows; the dashed line to `dim_employee` indicates it is not directly referenced by `fact_sales` via a foreign key but is part of the warehouse design.

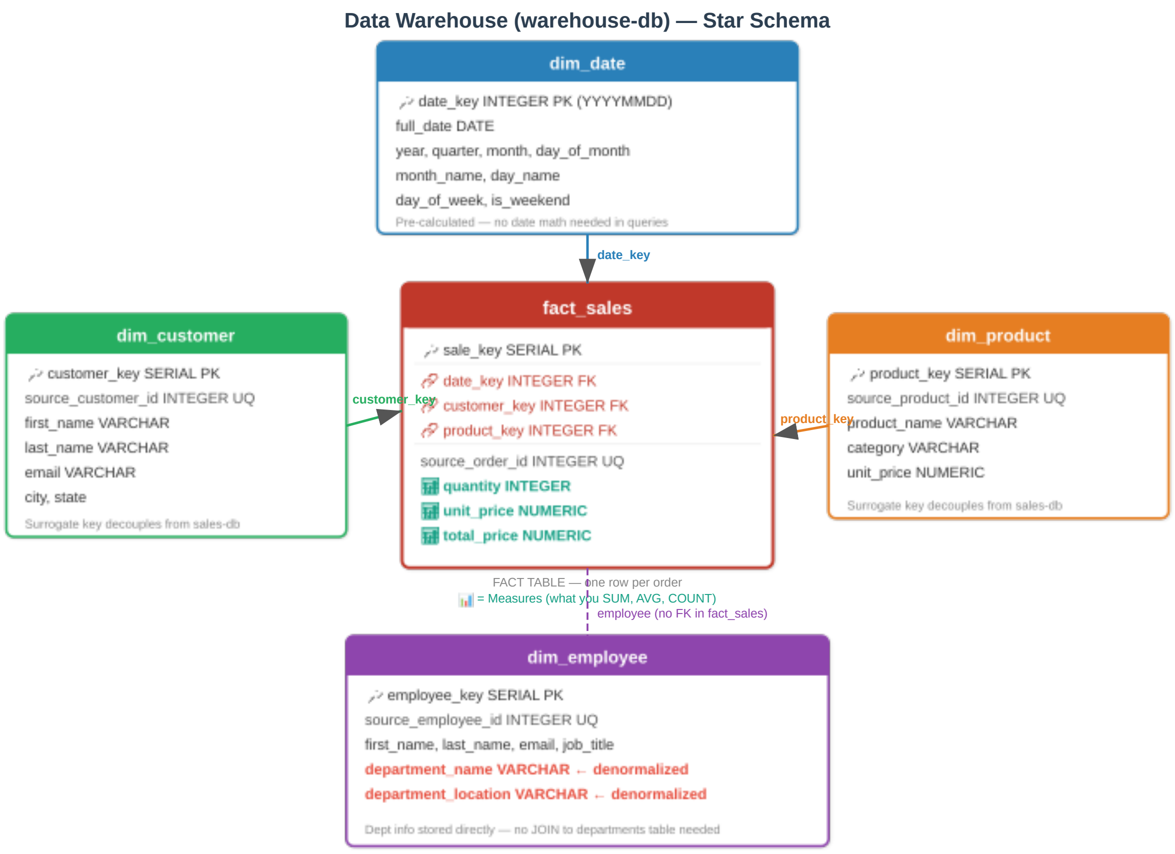

Data Warehouse (warehouse-db) — Star Schema
dim_date
date_key INTEGER PK (YYYYMMDD)
full_date DATE
year, quarter, month, day_of_month
month_name, day_name
day_of_week, is_weekend
Pre-calculated — no date math needed in queries
date_key
fact_sales
sale_key SERIAL PK
date_key INTEGER FK
customer_key INTEGER FK
product_key INTEGER FK
source_order_id INTEGER UQ
quantity INTEGER
unit_price NUMERIC
total_price NUMERIC
FACT TABLE — one row per order
📊 = Measures (what you SUM, AVG, COUNT)
dim_customer
customer_key SERIAL PK
source_customer_id INTEGER UQ
first_name VARCHAR
last_name VARCHAR
email VARCHAR
city, state
Surrogate key decouples from sales-db
customer_key
dim_product
product_key SERIAL PK
source_product_id INTEGER UQ
product_name VARCHAR
category VARCHAR
unit_price NUMERIC
Surrogate key decouples from sales-db
product_key
employee (no FK in fact_sales)
dim_employee
employee_key SERIAL PK
source_employee_id INTEGER UQ
first_name, last_name, email, job_title
department_name VARCHAR ← denormalized
department_location VARCHAR ← denormalized
Dept info stored directly — no JOIN to departments table needed

Figure 3: Data Warehouse (warehouse-db) — Star Schema ERD

The fact table sits at the center. Each dimension table connects to it through a foreign key. This is the "star" shape.

Why Stars Are Fast

Consider this business question: *"What was total revenue by product category and quarter in 2026?"*

In the warehouse (star schema) — one simple query:

```
SELECT p.category, d.quarter, SUM(f.total_price) AS revenue
FROM fact_sales f
JOIN dim_product p ON f.product_key = p.product_key
JOIN dim_date d ON f.date_key = d.date_key
WHERE d.year = 2026
```

```
GROUP BY p.category, d.quarter
ORDER BY d.quarter, revenue DESC;
```

In the operational database — more complex, slower:

```
SELECT p.category,
       EXTRACT(QUARTER FROM o.order_date) AS quarter,
       SUM(o.quantity * p.unit_price) AS revenue
FROM orders o
JOIN products p ON o.product_id = p.product_id
WHERE EXTRACT(YEAR FROM o.order_date) = 2026
GROUP BY p.category, EXTRACT(QUARTER FROM o.order_date)
ORDER BY quarter, revenue DESC;
```

The warehouse version is simpler (pre-calculated quarter, pre-computed total_price) and faster at scale (fewer rows to scan, indexed foreign keys, denormalized dimensions).

Surrogate Keys

You may have noticed that the dimension tables use two kinds of keys:

- **Surrogate key** (`customer_key`, `product_key`, etc.) — an auto-generated integer that serves as the primary key in the warehouse
- **Source key** (`source_customer_id`, `source_product_id`, etc.) — the original primary key from the operational database

Why not just use the operational database's key?

Surrogate keys provide several benefits:

1. **Stability**: If the operational system changes its key format (e.g., migrates from integer IDs to UUIDs), the warehouse is unaffected
2. **History tracking**: You can keep multiple versions of a customer record (e.g., before and after an address change) with different surrogate keys
3. **Cross-system integration**: If you load data from multiple operational systems, their keys might conflict; surrogate keys avoid collisions
4. **Performance**: Integer surrogate keys are compact and fast for joins

Other Common Warehouse Schemas

The star schema used in the midterm is the most common pattern, but you should be aware of two alternatives.

Snowflake Schema

A **snowflake schema** is a star schema where some dimension tables are further normalized into sub-tables. For example, instead of storing `department_name` directly in `dim_employee`, you might have a separate `dim_department` table that `dim_employee` references.

```
dim_department   dim_employee   fact_sales   dim_product   dim_category

                                   dim_date
```

Pros: Less data redundancy in dimensions; may save storage.

Cons: More JOINs needed for queries; more complex to understand and maintain.

In practice, star schemas are more common than snowflake schemas because the small amount of redundancy in dimension tables is a worthwhile trade-off for simpler, faster queries.

Galaxy Schema (Multi-Fact)

A **galaxy schema** (also called a fact constellation) has multiple fact tables that share some dimension tables. For example, a business might have both a `fact_sales` table and a `fact_returns` table, both referencing `dim_customer`, `dim_product`, and `dim_date`.

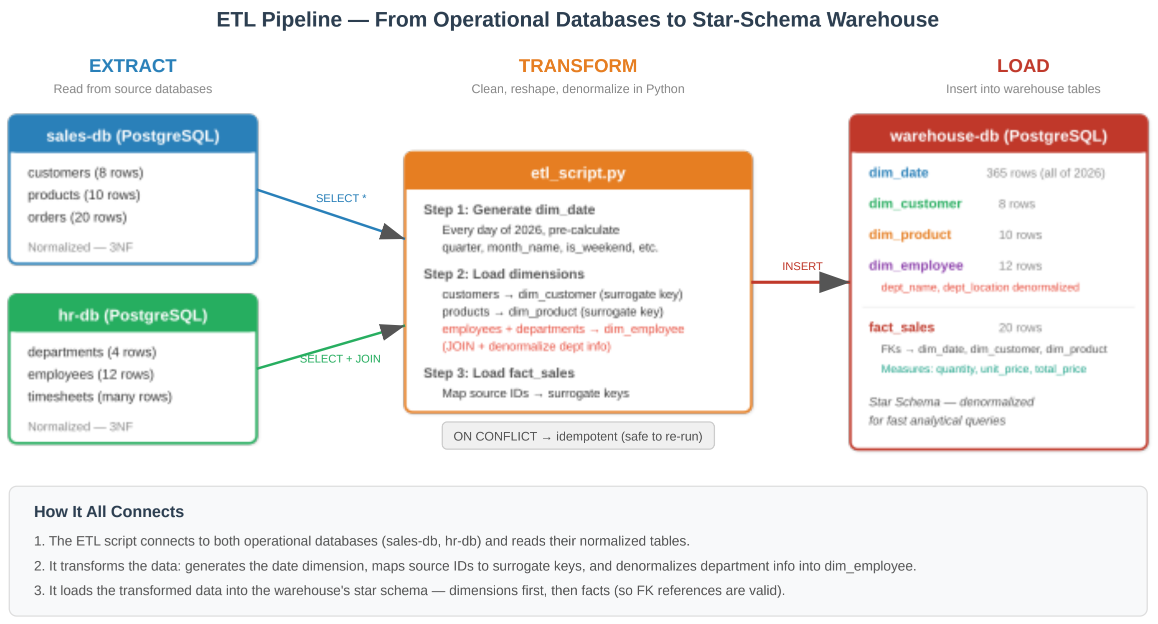
This is common in real-world warehouses but is beyond the scope of the midterm.

The ETL Pipeline

Data doesn't magically appear in the warehouse. An **ETL pipeline** moves data from operational databases to the warehouse:

1. **Extract**: Read data from operational databases (sales-db, hr-db)
2. **Transform**: Clean, reshape, and denormalize the data (e.g., join employees with departments, generate date dimension rows, compute derived fields)
3. **Load**: Insert the transformed data into the warehouse tables

The following diagram shows how the midterm's ETL pipeline moves data from the two operational databases into the star-schema warehouse:


ETL Pipeline — From Operational Databases to Star-Schema Warehouse
EXTRACT
Read from source databases
TRANSFORM
Clean, reshape, denormalize in Python
LOAD
Insert into warehouse tables
sales-db (PostgreSQL)
customers (8 rows)
products (10 rows)
orders (20 rows)
Normalized — 3NF
hr-db (PostgreSQL)
departments (4 rows)
employees (12 rows)
timesheets (many rows)
Normalized — 3NF
SELECT *
SELECT + JOIN
etl_script.py
Step 1: Generate dim_date
Every day of 2026, pre-calculate
quarter, month_name, is_weekend, etc.
Step 2: Load dimensions
customers → dim_customer (surrogate key)
products → dim_product (surrogate key)
employees + departments → dim_employee
(JOIN + denormalize dept info)
Step 3: Load fact_sales
Map source IDs → surrogate keys
ON CONFLICT → idempotent (safe to re-run)
INSERT
warehouse-db (PostgreSQL)
dim_date 365 rows (all of 2026)
dim_customer 8 rows
dim_product 10 rows
dim_employee 12 rows
dept_name, dept_location denormalized
fact_sales 20 rows
FKs → dim_date, dim_customer, dim_product
Measures: quantity, unit_price, total_price
Star Schema — denormalized
for fast analytical queries

How It All Connects

1. The ETL script connects to both operational databases (sales-db, hr-db) and reads their normalized tables.
2. It transforms the data: generates the date dimension, maps source IDs to surrogate keys, and denormalizes department info into dim_employee.
3. It loads the transformed data into the warehouse's star schema — dimensions first, then facts (so FK references are valid).

Figure 4: ETL Pipeline — From Operational Databases to Star-Schema Warehouse

In the midterm, the `etl_script.py` performs all three steps. Key design decisions in the midterm's ETL:

- **Idempotent loading**: The script uses `ON CONFLICT` clauses so it can be run multiple times without duplicating data. If a customer already exists, it updates the record; if an order already exists, it skips it.
- **Dimension loading before facts**: Dimensions are loaded first so that the foreign key references in `fact_sales` are valid.
- **Date dimension generation**: The date dimension is generated programmatically (every day of 2026) rather than extracted from an operational system.

Connecting to the Midterm

Here's how all of these concepts map to what you'll see in the midterm project:

Concept	Where You'll See It
Operational databases (OLTP)	`sales-db` and `hr-db` — normalized tables with JOINs
Analytical database (OLAP)	`warehouse-db` — star schema with denormalized dimensions
Fact table	`fact_sales` — one row per order, with measures (quantity, price)
Dimension tables	`dim_date`, `dim_customer`, `dim_product`, `dim_employee`
Denormalization	`dim_employee` includes `department_name` and `department_location` directly
Surrogate keys	`customer_key`, `product_key`, etc. (auto-generated, separate from source IDs)
Pre-calculated attributes	`dim_date` has `quarter`, `month_name`, `is_weekend` already computed
ETL pipeline	`etl_script.py` extracts from operational DBs, transforms, loads into warehouse
Idempotent loading	`ON CONFLICT` clauses in the load step prevent duplicate data

Key Takeaways

1. **Operational databases** are for recording transactions; **warehouses** are for analyzing them
2. **Normalization** (operational) eliminates redundancy; **denormalization** (warehouse) speeds up queries
3. A **star schema** has a central fact table (measures) surrounded by dimension tables (descriptive context)
4. **Fact tables** store what happened (quantities, amounts); **dimension tables** describe who, what, when, where
5. **Surrogate keys** decouple the warehouse from operational system changes
6. **ETL pipelines** move and reshape data from operational systems into the warehouse
7. The midterm uses all of these patterns — the supplemental readings and the assignment will walk you through each one

Further Reading

If you want to go deeper into data warehousing, these are good starting points:

- Ralph Kimball, *The Data Warehouse Toolkit* (the definitive reference for star schema design)

- Kimball Group Design Tips — short articles on specific design patterns
- Star Schema vs Snowflake Schema — IBM overview of schema patterns