

1 Normalization:

- Each fact is stored in exactly one place.
- Update, insert, and delete anomalies are avoided.
- Data integrity is easier to enforce (keys, foreign keys, constraints).
- Storage is used economically.

Normalization is the right answer to a particular question: *how do we keep data correct while many users read and modify it?* Today's question is different: *how do we make large-scale reading and analysis fast and cheap?* The answers can pull in opposite directions, which is why we study **denormalization**.

2 The Traditional World: Database Servers

The classic setting for normalized design is the **database server** (Oracle, PostgreSQL, MySQL, SQL Server).

- The server owns the data. Storage, indexes, buffer cache, and query processing are bundled in one system, often on one machine or a tightly coupled cluster.
- Workloads are mostly **OLTP**: many small transactions touching a few rows (place an order, update an address).
- Storage was expensive, so avoiding redundancy mattered.
- **Indexes** make a join cheap: look up a few matching rows instead of reading whole tables.
- **Joins** are the normal way to reassemble facts that normalization split apart.

In this world a **table scan** (reading every row) is a warning sign. The optimizer works hard to avoid scans by using indexes, and a query that scans a big table is usually considered a problem to be fixed.

3 What Changed

Several trends shifted the economics:

1. **Data volume.** Logs, clickstreams, sensor data, and event feeds reach terabytes and petabytes. Much of it is analyzed, not updated.

2. **Cheap storage.** Disk and object storage cost very little per gigabyte compared with the past. Redundant copies of a customer name are no longer a serious cost concern.
3. **Analytic workloads (OLAP).** Questions like “total revenue by region by month” touch millions or billions of rows but only a few columns.
4. **The cloud.** Organizations moved from owning servers to renting storage and compute by usage.

The key architectural consequence is the **separation of storage and compute**: data sits in one service, and processing power is rented separately, when needed, and scaled independently.

4 Cloud Storage and Flat Files

In the cloud, data commonly lives in **object storage** such as Amazon S3 (similar services: Azure Blob Storage, Google Cloud Storage).

- Data is stored as **files** (objects) in buckets, organized by path-like names, e.g. `s3://sales-data/orders`.
- Very cheap, very durable, and effectively unlimited in size.
- Objects are essentially **immutable**: you write new files rather than updating a row in place.
- There is no server, no buffer cache, and **no indexes**.

Common **flat file** formats:

- **CSV / TSV**: simple text, row oriented, easy but bulky and untyped.
- **JSON**: flexible and nested, but verbose.
- **Parquet, ORC**: compressed *columnar* binary formats with types and statistics, designed for analytics.

A flat file has no foreign keys and no enforced constraints. Nothing in the storage layer knows or cares that one file “references” another. Relationships are just a convention.

5 SQL Engines Instead of Database Servers

If the data is just files in storage, how do we query it? With a **SQL engine**: a system that provides SQL processing but does *not* own the data.

Examples: Amazon Athena, Presto/Trino, Apache Spark SQL, Hive, Google BigQuery, Snowflake.

	Database server	SQL engine
Owens the data?	Yes	No (reads files in place)
Storage	Internal, proprietary	Cloud object storage
Indexes	Central to performance	Usually none
Compute	Fixed server(s)	Elastic, scaled out
Typical workload	OLTP	OLAP / analytics
Updates	Row-level, transactional	Mostly append / rewrite files

OLTP (Online Transaction Processing) is optimized for fast, real-time transactional operations (e.g., banking, POS systems), while OLAP (Online Analytical Processing) is optimized for complex analytical queries on large, historical datasets using multidimensional models. OLTP ensures ACID-compliant updates to current data, whereas OLAP supports trend analysis, forecasting, and multidimensional reporting.

A table in a SQL engine is often just *metadata*: a schema plus the location of a set of files. For example:

```
CREATE EXTERNAL TABLE orders (  
  order_id    BIGINT,  
  customer_id BIGINT,  
  order_date  DATE,  
  total       DECIMAL(10,2)  
)  
STORED AS PARQUET  
LOCATION 's3://sales-data/orders/';
```

The engine spins up many workers, each reads a portion of the files, they cooperate on the result, and then the workers go away.

6 Table Scans Become the Normal Access Path

With no indexes, the default way to answer a query is to **scan** the files. In this environment that is not a failure; it is the design.

- The scan is **parallelized**: hundreds of workers each read a slice of the data at once.

- Sequential reads of large files are efficient, much more so than millions of random lookups against object storage.
- Techniques reduce how much is actually read:
 - **Columnar formats:** read only the columns the query uses.
 - **Partitioning:** organize files by a key such as date so that irrelevant folders are skipped (*partition pruning*).
 - **Compression:** fewer bytes to read, and repeated values compress extremely well.
 - **File statistics:** min/max values let the engine skip chunks of a file.

The mindset shift: in a database server we ask “how do I avoid scanning?” In a SQL engine we ask “how do I make each scan as small and as cheap as possible?”

7 Metered Compute and the Cost of Scans

Cloud services are **metered**: you pay for what you use. That makes performance a direct *financial* concern.

Two common billing models:

- **Pay per data scanned.** Athena and BigQuery’s on-demand mode charge by the amount of data a query reads (on the order of a few dollars per terabyte; check current pricing).
- **Pay per compute time.** Spark clusters and warehouses like Snowflake charge for the time workers are running.

In either model, **more data read and more work done means a bigger bill**. A poorly designed query or schema is not just slow, it is expensive, and every re-run costs money again.

Illustrative example (numbers are made up for illustration, assuming \$5 per TB scanned):

- A query that scans 2 TB costs about \$10.
- Run by a dashboard every 5 minutes, that is 288 runs a day, or about \$2,880 per day.
- Cutting the scan to 100 GB (columnar, partitioned) costs about \$0.50 per run, or about \$144 per day.

Design decisions that reduce bytes scanned and work done translate directly into savings.

8 The Problem with Joins at Scale

Normalization means answering most questions requires **joins**. On a single server with indexes, joins are fast. In a distributed SQL engine they are costly:

- Data for the two tables lives in different files on different workers.
- To match rows, the engine must **shuffle** data across the network so that matching keys land on the same worker. Network transfer is slow compared with local reads.
- Without indexes, a join typically means scanning *both* tables.
- **Data skew** (one customer or product with a huge share of rows) can overload a single worker and slow the whole query.
- Multi-way joins (orders, customers, products, regions, ...) multiply all of these costs.
- Joins consume memory and compute time, which are metered.

So the very mechanism normalization relies on to put the data back together is one of the most expensive operations in this environment.

9 Denormalization: Chunky Records

Denormalization means deliberately storing data in a less normalized form, combining related facts into one wide record (sometimes called a *chunky* or *wide* record), to make reads faster and cheaper.

Normalized design:

```
orders(order_id, customer_id, product_id, order_date, quantity)
customers(customer_id, name, city, state, segment)
products(product_id, product_name, category, price)
```

A report of revenue by state and category requires joining all three tables.

Denormalized design:

```
orders_wide(order_id, order_date, quantity,
            customer_id, customer_name, city, state, segment,
            product_id, product_name, category, price)
```

The same report is now a single scan with a filter and a GROUP BY:

```
SELECT state, category, SUM(quantity * price) AS revenue
FROM orders_wide
WHERE order_date >= DATE '2026-01-01'
GROUP BY state, category;
```

Denormalized records can also hold nested or repeated structures (e.g. a JSON order with an array of line items inside it), so that everything about one business event lives in one record.

9.1 Motivation

- **Avoid joins**, and with them the shuffles, skew, and extra scans.
- **One pass over the data**: simpler, faster execution plans.
- **Lower metered cost**: less compute time, fewer repeated operations.
- **Simpler queries** for analysts: no need to know how to join the schema correctly.
- **Redundancy is cheap**: storage costs little, and columnar compression shrinks repeated values dramatically. Because a columnar engine reads only the columns it needs, extra columns that a query does not use cost almost nothing at query time.

9.2 Trade-offs

Denormalization is not free. The problems normalization solved are still there:

- **Redundancy**: a customer's city is repeated on every order.
- **Update anomalies**: if the city changes, many records are stale or must be rewritten.
- **Inconsistency risk**: copies can disagree if not managed carefully.
- **Larger raw size** and wider records.

Why it is acceptable in this setting:

- Analytic data is mostly **append-only**. An order that happened in October stays an order that happened in October.
- Often the redundancy is *correct*: the customer's city *at the time of the order* is exactly what a historical report wants.
- Data is typically **rebuilt or reloaded** in batches by a pipeline, so “updating” means regenerating files, not editing rows in place.

10 A Common Architecture

In practice, normalization and denormalization are used together, in different places:

1. An **operational database** (normalized, OLTP) is the system of record where transactions happen and integrity is enforced.
2. A **pipeline** (ETL/ELT) extracts data, joins it once, and writes denormalized files, often Parquet partitioned by date, into cloud storage.
3. **SQL engines** query the wide tables for reports, dashboards, and data science.

The join is paid for *once* at load time rather than every time someone runs a query. This is the core trade: **do the expensive work once, write the result, and read it cheaply many times.**

11 Why Denormalization Makes More Sense Here

Normalization assumed...	Cloud / big data reality
Storage is expensive	Storage is cheap; compute is the metered cost
Indexes make joins fast	Usually no indexes; joins mean shuffles
Single server owns the data	Data in files; compute is distributed
Frequent row-level updates	Mostly append-only, batch loaded
Scans are to be avoided	Parallel scans are the normal access path
Integrity enforced by the DBMS	Integrity handled upstream in the pipeline

Normalization optimizes for *storage space and update correctness*. In the cloud analytics setting, what we pay for is *reading data and computing*, so the design that minimizes reads and joins wins, even at the cost of redundancy.

12 When Not to Denormalize

- **Transactional systems** with frequent updates and strict integrity needs. Keep them normalized.
- Data that **changes often** and must be consistent everywhere (a current price, a current address).
- **Very large dimensions** that are rarely used and would bloat every record.
- When you need a **single source of truth** and the pipeline to rebuild copies does not exist.

Denormalization is a *design choice driven by the workload*, not a rule that replaces normalization.

13 Key Takeaways

1. Normalization was designed for correctness and economical storage on database servers with indexes.
2. The move to the cloud separated storage (cheap files in S3) from compute (elastic SQL engines).
3. Without indexes, table scans are the default, and cloud billing makes scanned bytes and compute time cost real money.
4. Joins at scale require shuffling data across the network and are expensive.
5. Denormalized, chunky records avoid joins, reduce scans and cost, and simplify queries.
6. The price is redundancy and potential inconsistency, which is manageable for append-only analytic data built by pipelines.
7. Use both: normalized systems of record, denormalized analytic copies.