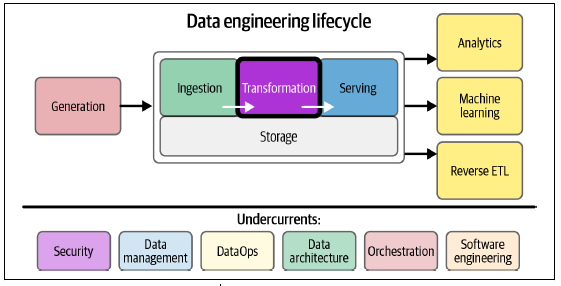
# Fundamentals of Data Engineering - Reis & Housley

## Part II. The Data Engineering Lifecyle in Depth

### Chapter 8 – Queries, Modeling, and Transformation

* Up to this point, the stages of the DE lifecycle have primarily been about passing data from one place to another or storing it
* Next, you’ll learn how to make data *useful*
* **By understanding queries, modeling, + transformations, you’ll have the tools to turn raw data ingredients into something consumable by downstream stakeholders, since transformations allow us to create *value* from data**
* We first discuss **queries** + the **significant patterns underlying them**
* Then, we look at the **major data modeling patterns you can use to introduce business logic into your data**
* Then, we cover **transformations**, which take the logic of data models + the results of queries and make them useful for more straightforward downstream consumption
* Finally, we cover whom you’ll work with + the undercurrents as they relate to the above topics



* A variety of techniques can be used to query, model, and transform data in SQL + NoSQL databases
* We **focus on queries made to an *OLAP* system, such as a DW or data lake**
* Although many languages exist for querying, for the sake of convenience + familiarity, we focus heavily on **SQL**, the most popular + universal query language
* *Most concepts for OLAP databases + SQL will translate to other types of databases + query languages*
* You need an understanding of the SQL language + related concepts like primary and foreign keys
* For convenience, we’ll use the term “database” as a shorthand for a query engine *and* the storage it’s querying (this could be a cloud DW or Apache Spark querying data stored in S3)
* We assume the database has a storage engine that organizes the data under the hood
* This extends to file-based queries (loading a CSV file into a Python notebook) + queries against file formats such as Parquet
* Also, note we **focus mainly on the query, modeling patterns, + transformations related to** **structured** and **semi-structured** data, which **DE’s use often**
* Many of the practices discussed can also be applied to working w/ **unstructured** data such as images, video, + raw text

#### Queries

* **Queries** are a fundamental part of DE, data science, + analysis
* **Before you learn about the underlying patterns + tech for transformations, you need to understand what queries *are*, *how* they work on various data, + techniques for *improving* query *performance***
* This section primarily concerns itself with queries on **tabular** + **semi-structured data**
* As a DE, you’ll most frequently query + transform these data types

##### What is a Query?

* We often run into people who know how to write SQL but are **unfamiliar w/ how a query works under the hood**
* Some of this introductory material on queries will be familiar to experienced DE’s, so feel free to skip ahead if this applies to you
* A **query** allows you to **retrieve + act on data**
* Remember CRUD (Chapter 5) 🡪 When a query *retrieves* data, it is issuing a *request* to ***read*** (R) a pattern of records
* You might issue a query that gets all records from a table foo, such as *SELECT \* FROM foo*
* Or you might apply a **predicate** **(logical condition)** to filter data by retrieving only records where the id is 1, using the SQL query *SELECT \* FROM foo WHERE id=1*
* Many databases allow you to **create**, **update**, and **delete** data (CUDin CRUD)
* **Such queries will either create, mutate, or destroy existing records**
* Let’s review some other common acronyms you’ll run into when working w/ query languages

###### a) Data Definition Language

* At a high level, you first need to *create* the database objects before adding data via **data definition language (DDL)** commands to **perform operations on database objects, such as the database itself, schemas, tables, or users**
* **DDL defines the state of objects in a database**
* DE’s use common SQL DDL expressions: CREATE, DROP, and UPDATE
* Ex: You can create a database by using the DDL expression *CREATE DATABASE ‘bar’*
* After, you can create new tables (*CREATE table bar\_table*) or delete a table (*DROP table bar\_table*)

###### b) Data Manipulation Language

* After using DDL to define database objects, you need to add + alter data w/in such objects, which is the primary purpose of **data manipulation language (DML)**
* Some common DML commands you’ll use as a DE are: **SELECT, INSERT, UPDATE, DELETE, COPY, MERGE**
* Ex: You can INSERT new records into a database table, UPDATE existing ones, + SELECT specific records

###### c) Data Control Language

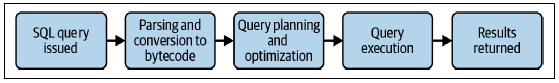
* You most likely want to **limit access to database objects and finely control *who* has access to *what***
* **Data control language (DCL)** **allows you to control access to the database objects or the data by using SQL commands** such as **GRANT, DENY, + REVOKE**
* Let’s walk through a brief example using DCL commands
* A new data scientist named Sarah joins your company, + she needs read-only access to a database called *data\_science\_db*
* You give Sarah access to this database by using the following DCL command:
* **GRANT SELECT ON** data\_science\_db **TO** user\_name Sarah;
* It’s a hot job market, + Sarah has worked at the company for only a few months before getting poached by a big tech company
* Being a security-minded DE, you remove Sarah’s ability to read from the database:
* **REVOKE SELECT ON** data\_science\_db **TO** user\_name Sarah;
* **Access-control requests + issues are common, so understanding DCL will help you resolve problems if you or a team member can’t access the data they need, as well as prevent access to data they don’t need**

###### Transaction Control Language

* As its name suggests, **transaction control language (TCL)** supports commands that **control the details of transactions**
* W/ TCL, **we can define commit checkpoints, conditions when actions will be rolled back, + more**
* 2 common TCL commands include **COMMIT** and **ROLLBACK**

##### The Life of a Query

* *How does a query work, + what happens when a query is executed?*
* We can cover the high-level basics of query execution using an example of a typical SQL query executing in a database
* While running a query might seem simple (write code, run it, + get results), a lot is going on under the hood
* When you execute a SQL query, here’s a summary of what happens:



* **1. The database engine** **compiles the SQL**, parsing the code to **check for proper semantics** + **ensuring that the database objects referenced exist** + that the **current user has the appropriate access to these objects**
* **2. The SQL code is converted into bytecode**
* This bytecode **expresses the steps that must be executed** on the database engine in an **efficient, machine-readable format**
* **3. The database’s query optimizer analyzes the bytecode to determine how to execute** the query, **reordering + refactoring steps to use available resources as efficiently as possible**
* **4.** The query is **executed**, and **results are produced**

##### The Query Optimizer

* **Queries can have wildly different execution times, *depending on how they’re executed***
* A **query optimizer’s job is to optimize query performance + minimize costs by breaking the query into appropriate steps in an efficient order**
* The optimizer will **assess joins, indexes, data scan size, + other factors** as it attempts to execute the query in the least expensive manner
* **Query optimizers are fundamental to how your query will perform**
* **Every database is different + executes queries in ways that are obviously + subtly different from each other**
* You **won’t *directly* work w/ a query optimizer**, but **understanding some of its functionality will help you write more performant queries**
* **Need to know how to analyze a query’s performance**, using things like an **explain plan** or **query analysis** (described next)

##### Improving Query Performance

* In DE, you’ll **inevitably encounter poorly performing queries**
* **Knowing how to identify + fix these poor queries is invaluable**
* **Don’t fight your database, + learn to work w/ its strengths + augment its weaknesses**
* There are **various ways to improve your query performance**

###### 1) Optimize Join Strategy and Schema

* A single dataset (such as a table or file) is rarely useful on its own, + we **create value by combining it with other datasets**
* **Joins**are **one of the most common means of combining datasets + creating new ones**
* **Significant types of joins = INNER, OUTER, LEFT/RIGHT, CROSS**
* **Types of join relationships = one to one, one to many, many to one, and many to many**
* **Joins are critical in DE, and are well supported + performant in many databases**
* **Even columnar databases, which in the past had a reputation for slow join performance, now generally offer excellent performance**
* A **common technique for improving query performance is to pre-joindata**
* If you find that **analytics queries are joining the same data repeatedly, it often makes sense to join the data *in advance* + have queries read from the pre-joined version of the data so that you’re not repeating computationally intensive work**
* This **may mean changing the schema + relaxing normalization conditions to widen tables and utilize newer data structures (such as arrays or structs) for replacing frequently joined entity relationships**
* **Another strategy** is **maintaining a more normalized schema but pre-joining tables for the most common analytics + data science use cases**
* We **can simply create pre-joined tables + train users to utilize these or join inside materialized views**
* Next, **consider the details + complexity of your join conditions**
* **Complex join logic may consume significant computational resources**
* We **can improve performance for complex joins in a few ways**
* **Many row-oriented databases allow you to index a result computed from a row**
* Ex: PostgreSQL allows you to create an index on a string field converted to lowercase
* When the optimizer encounters a query where the lower() function appears inside a predicate, it can apply the index
* You can also create a new derived column for joining, though you will need to train users to join on this column.
* Finally, **use common table expressions (CTEs)** **instead of nested subqueries or temp tables**
* CTEs **allow users to compose complex queries together in a readable fashion, helping you understand the flow of your query**
* **The importance of readability for complex queries cannot be understated.**
* **In many cases, CTEs will also deliver better performance than a script that creates intermediate tables**
* If you *have* to create intermediate tables, consider creating temp tables
* **NOTE: Row Explosion**
* An obscure but frustrating problem is **row explosion**, which **occurs when we have a large number of many-to-many matches, either b/c of repetition in join keys or as a consequence of join logic**
* Suppose the join key in table A has the value “this” repeated 5 times, + the join key in table B contains this *same* value repeated 10 times
* This leads to a **cross-join** of these rows: every “this” row from table A paired w/ every “this” row from table B, which creates 5 × 10 = 50 rows in the output
* Now suppose that many other repeats are in the join key
* **Row explosion often generates enough rows to consume a massive quantity of database resources or even cause a query to fail**
* It is **also essential to know how your query optimizer handles joins**
* ***Some databases can reorder joins + predicates, while others cannot***
* A row explosion in an early query stage may cause the query to fail, even though a later predicate should correctly remove many of the repeats in the output
* **Predicate reordering can significantly reduce the computational resources required by a query**

###### 2) Use the Explain Plan and Understand Your Query’s Performance

* As noted, **the database’s query optimizer influences the execution of a query**
* The **query optimizer’s explain plan will show you how the optimizer determined its optimum lowest-cost query, the database objects used (tables, indexes, cache, etc.), + various resource consumption + performance statistics in each query stage**
* **Some databases provide a visual representation** of query stages, while **others make the explain plan available via SQL with the EXPLAIN command**, which displays the sequence of steps the database will take to execute the query
* **In addition to using EXPLAIN to understand *how* a query will run, you should monitor a query’s performance, viewing metrics on database resource consumption.**
* The following are some areas to monitor:
* **Usage of key resources** such as disk, memory, and network.
* **Data loading time vs. processing time**
* **Query execution time**, **number of records**, the **size of the data scanned**, + the **quantity of data shuffled**
* **Competing queries** that might cause **resource contention** in your database.
* **Number of concurrent connections** used vs. **connections available**
* **Oversubscribed concurrent connections can have negative effects** on your users who may not be able to connect to the database

###### 3) Avoid Full Table Scans

* **All queries scan data, but not all scans are created equal**
* Rule of thumb 🡪 you should **query only the data you need**
* When you run **SELECT \* w/ no predicates**, you’re **scanning the entire table**, retrieving every row + column
* This is **very inefficient performance-wise *and* expensive**, especially if using a pay-as-you-go database that charges you either for bytes scanned or compute resources utilized while a query is running
* **Whenever possible, use pruningto reduce the quantity of data scanned in a query**
* **Columnar and row-oriented databases require different pruning strategies**
* In a **column-oriented database**, you should **select only the columns you need**
* Most column-oriented OLAP databases also provide additional tools for optimizing tables for better query performance
* Ex: If you have a very large table (several TB in size or greater), Snowflake and BigQuery give you the option to define a **cluster key** on a table, which **orders the table’s data in a way that allows queries to more efficiently access portions of very large datasets**
* BigQuery also allows you to **partition** **a table into smaller segments, allowing you to query only specific partitions instead of the entire table**
* **Be aware that inappropriate clustering + key distribution strategies can degrade performance**
* In **row-oriented databases, pruning usually centers around table indexes** (see Chapter 6)
* **Indexes** **provide a map of the table for particular fields + allow extremely fast lookup of individual records**
* **W/out indexes, a database would need to scan an *entire* table to find the records satisfying a WHERE condition**
* **General index strategy = Create table indexes that will improve performance for your *most performance-sensitive queries* while NOT overloading the table w/ so many indexes such that you degrade performance**

###### 4) Know How Your Database Handles Commits

* **A database commitis a change w/in a database, such as creating, updating, or deleting a record, table, or other database objects**
* **Many databases support transactions** (i.e., a notion of **committing *several* operations simultaneously in a way that maintains a consistent state**)
* ***Please note that the term “transaction“ is somewhat overloaded (see Chapter 5)***
* The **purpose of a transaction is to keep a consistent state of a database both while it’s active + in the event of a failure**
* **Transactions also handle isolation when multiple concurrent events might be reading, writing, + deleting from the same database objects**
* **W/out transactions, users would get potentially conflicting information when querying a database**
* You should **be intimately familiar with how your database handles commits *and* transactions, and determine the expected consistency of query results**
* Does your **database handle writes + updates in an ACID-compliant manner?**
* ***W/out* ACID (atomicity, consistency, isolation, durability) compliance, a query might return unexpected results**
* Could result from a **dirty read**, which happens **when a row is read and an uncommitted transaction has altered the row**
* *Are dirty reads an expected behavior of your database?*
* *If so, how do you handle this?*
* Also, **be aware that during UPDATE + DELETE transactions, some databases create new files to represent the new state of the database + retain the old files for failure checkpoint references**
* **In *these* databases, running a large number of small commits can lead to clutter and consume significant storage space that might need to be vacuumed periodically**
* Briefly consider 3 databases to understand the impact of commits (current as of 2022-2023)
* 1) Suppose we’re looking at a **PostgreSQL RDBMS and are applying ACID transactions**
* **Each transaction consists of a *package* of operations that will either fail or succeed *as a group***
* **Can also run analytics queries across many rows**, + these queries will present a **consistent picture of the database at a point in time**
* The ***disadvantage*** of the PostgreSQL approach is that it **requires row locking(blocking reads and writes to certain rows), which can degrade performance in various ways**
* **PostgreSQL is NOT optimized for large scans or the massive amounts of data appropriate for large-scale analytics applications**
* 2**) Google BigQuery** utilizes a **point-in-time full table commit model**
* When a read query is issued, **BigQuery reads from the latest committed snapshot of a table**
* Whether the query runs for 1 second or 2 hours, it **will read *only* from that snapshot and will *not* see any subsequent changes**
* **BigQuery does NOT lock the table while reading from it**
* Instead, **subsequent write operations will create new commits + new snapshots while the *query continues to run on the snapshot where it started***
* **To prevent the inconsistent state, BigQuery allows only one write operation at a time**
* **In this sense, BigQuery *provides no write concurrency what-so-ever***
* In the sense that **it can write massive amounts of data in parallel *inside a single write query*, it *is* highly concurrent**
* **If more than one client attempts to write simultaneously, write queries are queued in order of arrival**
* BigQuery’s commit model is similar to the commit models used by Snowflake, Spark, + others
* 3) **MongoDB** is a **variable-consistency database**
* DE’s have **various configurable consistency options, both for the database *and* at the level of individual queries**
* MongoDB is celebrated for its **extraordinary scalability + write concurrency** but is somewhat **notorious for issues that arise when engineers abuse it**
* Ex: In certain modes, MongoDB supports **ultra-high write performance.**
* However, this **comes at a cost**: the **database will unceremoniously + silently discard writes if it gets overwhelmed w/ traffic**
* This is perfectly **suitable for applications that can stand to lose some data** (say, IoT applications where we simply want many measurements but don’t care about capturing *all* measurements)
* It is **NOT a great fit for applications that need to capture *exact* data + statistics**
* None of this is to say these are bad databases
* They’re all **fantastic databases *when chosen for appropriate applications + configured correctly***
* The **same goes for virtually any database technology**
* Companies don’t hire DE’s simply to hack on code in isolation
* To be worthy of their title, **DE’s should develop a deep understanding of the problems they’re tasked w/ solving + the technology tools**
* This **applies to commit + consistency models and every other aspect of technology performance**
* **Appropriate technology choices + configuration can ultimately differentiate extraordinary success and massive failure** (see Chapter 6 for a deeper discussion of consistency)

###### 5) Vacuum Dead Records

* As just discussed, **transactions incur the overhead of creating new records during certain operations, such as updates, deletes, + index operations, while retaining the old records as pointers to the last state of the database**
* **As these old records accumulate in the database filesystem, they eventually no longer need to be referenced**
* You should **remove these dead records in a process called vacuuming**
* You **can vacuum a single table, multiple tables, or *all* tables in a database**
* No matter *how* you choose to vacuum, **deleting dead database records is important for a few reasons:**
* **1) It frees up space for new records, leading to less table bloat + faster queries**
* **2) New *and relevant* records mean query plans are more accurate**
* *Outdated records can lead a query optimizer to generate suboptimal + inaccurate plans*
* **3) Vacuuming cleans up poor indexes, allowing for better index performance**
* **Vacuum operations are handled differently depending on the type of database**
* Ex: **In databases backed by object storage** (BigQuery, Snowflake, Databricks), the only downside of **old data retention is that it uses storage space, potentially costing money** depending on the storage pricing model for the database
* Ex: In Snowflake, users cannot *directly* vacuum
* Instead, they control a “time-travel” interval that determines how long table snapshots are retained before they are auto vacuumed.
* Ex: BigQuery utilizes a fixed 7-day history window
* Ex: Databricks generally retains data indefinitely until it is *manually* vacuumed
* Vacuuming is important to control direct S3 storage costs
* Ex: Amazon Redshift handles its cluster disks in many configurations, + **vacuuming can impact performance + available storage**
* VACUUM runs automatically behind the scenes, but users may sometimes want to run it manually for tuning purposes
* **Vacuuming becomes even more critical for relational databases** such as PostgreSQL and MySQL
* **Large numbers of transactional operations can cause a rapid accumulation of dead records**
* **DE’s working in these systems need to familiarize themselves w/ the details + impact of vacuuming**

###### 6) Leverage Cached Query Results

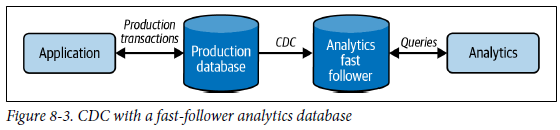
* Let’s say you have an **intensive query that you often run on a database that charges you for the amount of data you query**
* **Each time a query is run, this costs you money**
* Instead of rerunning the *same* query on the database repeatedly + incurring massive charges, **wouldn’t it be nice if the results of the query were stored + available for instant retrieval?**
* Thankfully, **many cloud OLAP databases cache query results**
* When a query is *initially* run, it will retrieve data from various sources, filter + join it, and output a result
* This initial query (a **cold query**) is similar to the notion of **cold data (infrequently accessed data)** explored in Chapter 6
* For argument’s sake, let’s say this query took 40 seconds to run
* Assuming your database caches query results, rerunning the same query might return results in 1 second or less
* The results were cached, + the query didn’t need to run cold
* **Whenever possible, leverage query cache results to reduce pressure on a database + provide a better user experience for frequently run queries**
* Note also that **materialized viewsprovide another form of query caching** (discussed later)

##### Queries on Streaming Data

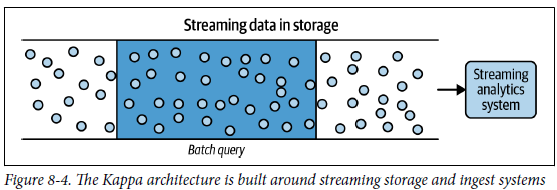
* **Streaming data is constantly in flight**, + querying streaming data is different from batch data
* To fully take advantage of a data stream, we **must adapt query patterns that reflect its real-time nature**
* Ex: Systems such as Kafka and Pulsar make it easier to query streaming data sources

###### a) Basic Query Patterns on Streams

* Recall **continuous CDC** (Chapter 7)
* CDC, in this form, **essentially sets up an analytics database as a fast follower to a production database**
* **One of the longest-standing streaming query patterns simply entails querying the analytics database, retrieving statistical results + aggregations w/ a slight lag behind the production database**
* **The Fast-Follower Approach**
* How is this a streaming query pattern? Couldn’t we accomplish the same thing simply by running our queries on the production database? 🡪 *In principle, yes; in practice, no*
* **Production databases generally aren’t equipped to handle production workloads + simultaneously run large analytics scans across significant quantities of data**
* Running such queries can slow the production application or even cause it to crash
* The **basic CDC query pattern allows us to serve real-time analytics w/ a minimal impact on the production system**
* The fast-follower pattern ***can* utilize a conventional transactional database as the follower, but there are significant advantages to using a proper OLAP-oriented system**



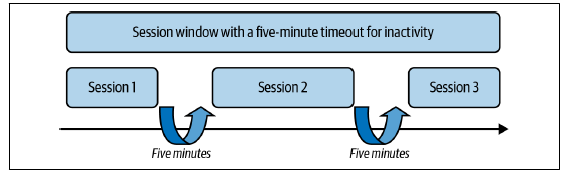
* Both Druid + BigQuery **combine a streaming buffer w/ long-term columnar storag**e in a setup **somewhat** **similar to the Lambda architecture** (Chapter 3)
* This **works extremely well for computing trailing statistics on vast historical data w/ near real-time updates.**
* The **fast-follower CDC approach has critical limitations**
* It **doesn’t fundamentally rethink batch query patterns**, and you’re **still running SELECT queries against the current table state** + **missing the opportunity to dynamically trigger events off changes in the stream**
* **The Kappa architecture**
* Next, recall the Kappa architecture (Chapter 3), where **the principal idea is to handle all data like events and store these events as a stream rather than a table**



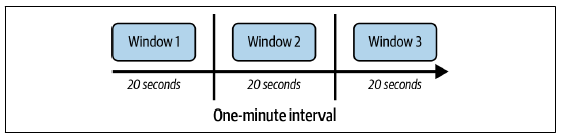
* **When production application databases are the source, Kappa architecture stores events from CDC**
* **Event streams can also flow directly from an application backend, from a swarm of IoT devices, or any system that generates events and can push them over a network**
* Instead of simply treating a streaming storage system as a buffer, **Kappa architecture retains events in storage during a more extended retention period, + data can be directly queried from this storage**
* The retention period can be pretty long (months or years)
* Note that this is much longer than the retention period used in purely real-time oriented systems, usually a week at most
* The **“big idea” in Kappa architecture is to treat streaming storage as a real-time transport layer and a database for retrieving + querying historical data**
* This **happens either through the direct query capabilities of the streaming storage system or with the help of external tools**
* Ex: Kafka KSQL supports aggregation, statistical calculations, + even **sessionization**
* If query requirements are more complex or data needs to be combined w/ other data sources, an external tool such as Spark reads a time range of data from Kafka + computes the query results
* The streaming storage system can also feed other applications or a stream processor such as Flink or Beam

###### b) Windows, Triggers, Emitted Statistics, and Late-Arriving Data

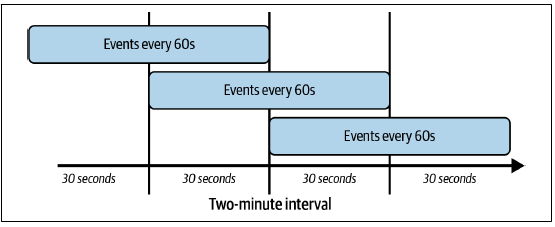
* **One fundamental limitation of traditional batch queries is that this paradigm generally treats the query engine as an external observer**
* An **actor external to the data causes the query to run**, perhaps an hourly CRON job or a product manager opening a dashboard
* **Most widely used streaming systems, on the other hand, support the notion of computations triggered directly from the data itself**
* They might emit mean + median statistics every time a certain number of records are collected in the buffer or output a summary when a user session closes
* **Windows are an essential feature in streaming queries and processing**
* Windows are **small batches that are processed based on dynamic triggers** + are **generated dynamically over time in some ways**
* Let’s look at some common types of windows: **session**, **fixed-time**, and **sliding**
* We’ll also look at **watermarks**.
* **Session window**
* A session window **groups events that occur close together, + filters out periods of inactivity when no events occur**
* We might say that a “user session” is any time interval w/ no inactivity gap of 5 minutes or more
* The batch system collects data by a user ID key, orders events, determines the gaps and session boundaries, + calculates statistics for each session
* DE’s **often** **sessionize data retrospectively by applying time conditions to user activity on web + desktop apps**
* **But in a streaming session, this process can happen dynamically**
* Note that session windows are *per key*
* In the preceding example, *each user gets their own set of windows*
* The system accumulates data per user
* If a 5-minute gap w/ no activity occurs, the system closes the window, sends its calculations, + flushes the data
* If new events arrive for the user, the system starts a new session window
* **Session windows may also make a provision for late-arriving data**
* Allowing data to arrive up to 5 minutes late to account for network conditions and system latency, the system will open the window if a late-arriving event indicates activity < 5 minutes after the last event
* Below shows 3 session windows, each separated by 5 minutes of inactivity



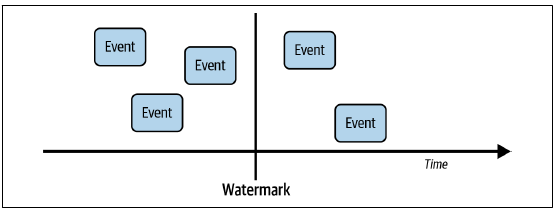
* **Making sessionization dynamic + near real-time fundamentally changes its utility**
* W/ *retrospective* sessionization, we could automate specific actions a day or an hour after a user session closed (e.g., a follow-up email w/ a coupon for a product viewed by the user)
* W/ ***dynamic* sessionization**, the same user could get an alert in a mobile app that is immediately useful based on their activity in the last 15 minutes
* **Fixed-time windows**
* A fixed-time (aka tumbling) window **features fixed time periods that run on a fixed schedule and processes all data since the previous window is closed**
* Ex: Might close a window every 20 seconds + process all data arriving from the previous window to give a mean + median statistic



* **Statistics would be emitted as soon as they could be calculated after the window closed**
* This is **similar to traditional batch ETL processing**, where we might run a data update job every day or every hour
* The **streaming system allows us to generate windows more frequently + deliver results w/ lower latency**
* As emphasized, ***batch is a special case of streaming***
* **Sliding windows**
* **Events** in a sliding window are **bucketed into windows of fixed time length, where separate windows might overlap**
* Ex: We could generate a new 60-second window every 30 seconds



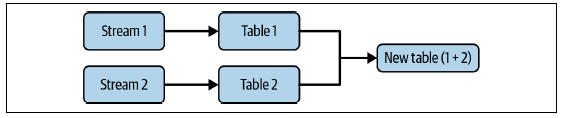
* Just as before, we can emit mean + median statistics
* The **sliding can vary**
* Ex: We might think of the window as truly sliding continuously but emitting statistics only when certain conditions (triggers) are met
* Suppose we used a 30-second continuously sliding window but calculated a statistic only when a user clicked a particular banner
* This would lead to an extremely high rate of output when many users click the banner, + no calculations during a lull
* **Watermarks**
* **Data is sometimes ingested out of the order from which it originated**
* A **watermark** is **a threshold used by a window to determine whether data in a window is w/in the established time interval or whether it’s considered late**



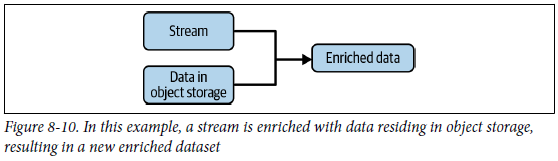
* **If data arrives that is new to the window but older than the timestamp of the watermark, it is considered to be late-arriving data**

###### c) Combining Streams with Other Data

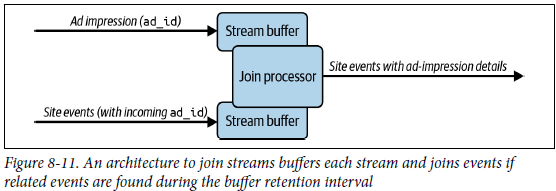
* As mentioned before, we **often derive value from data by combining it w/ other data**, + streaming data is no different
* For instance, **multiple streams can be combined, or a stream can be combined w/ batch historical data**
* Conventional table joins
* **Some tables may be fed by streams**
* The **most basic approach to this problem is simply joining these 2 tables in a database**
* A **stream can feed one or both of these tables**



* Enrichment
* **Enrichment means that we join a stream to other data**



* **Typically**, this is **done to provide enhanced data into *another* stream**
* Ex: Suppose an online retailer receives an event stream from a partner business containing product + user IDs
* The retailer wishes to enhance these events w/ product details + demographic information on the users
* The retailer feeds these events to a serverless function that looks up the product and user in an in-memory database (say, a cache), adds the required information to the event, + outputs the enhanced events to another stream.
* In practice, the **enrichment source could originate almost anywhere** (a table in a cloud DW or RDBMS, or a file in object storage)
* It’s simply a question of reading from the source + storing the requisite enrichment data in an appropriate place for retrieval by the stream
* Stream-to-stream joining
* **Increasingly, streaming systems support direct stream-to-stream joining**
* Suppose an online retailer wishes to join its web event data w/ streaming data from an ad platform
* The company can feed both streams into Spark, but a variety of complications arise
* Ex: The streams may have significantly different latencies for arrival at the point where the join is handled in the streaming system
* The ad platform may provide its data with a 5-minute delay
* In addition, certain events may be significantly delayed
* Ex: A session close event for a user, or an event that happens on the phone offline + shows up in the stream only after the user is back in mobile network range
* As such, **typical streaming join architectures rely on streaming buffers**
* The **buffer retention interval is configurable**, + a longer retention interval requires more storage + other resources
* Events get joined w/ data in the buffer + are eventually evicted after the retention interval has passed



#### Data Modeling

* Data modeling is something overlooked disturbingly often
* It can be common for data teams to jump into building data systems w/out a game plan to organize their data in a way that’s useful for the business, which is a mistake
* **Well-constructed data architectures must reflect the goals + business logic of the organization that relies on this data**
* **Data modeling involves deliberately choosing a coherent structure for data + is a critical step to make data useful for the business**
* It has been a practice for decades in one form or another
* Ex: Various types of normalization techniques have been used to model data since the early days of RDBMSs, + DW modeling techniques have been around since at least the early 1990s and arguably longer
* As pendulums in technology often go, data modeling became somewhat unfashionable in the early to mid-2010s
* The rise of data lake 1.0, NoSQL, and big data systems allowed DE’s to bypass traditional data modeling, *sometimes for legitimate performance gains*
* Other times, the **lack of rigorous data modeling created data swamps, along with lots of redundant, mismatched, or simply wrong data**
* **Nowadays, the pendulum seems to be swinging back toward data modeling**
* The **growing popularity of data management (in particular, data governance + data quality) is pushing the need for coherent business logic**
* The meteoric rise of data’s prominence in companies creates a growing recognition that **modeling is critical for realizing value** at the higher levels of the Data Science Hierarchy of Needs pyramid
* That said, ***new* paradigms are required to truly embrace the needs of streaming data and ML**

##### What is a Data Model?

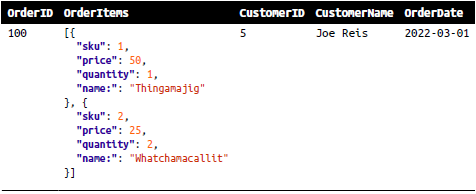
* A **data model****represents the way data relates to the real world**
* It **reflects how the data must be structured and standardized to best reflect your organization’s processes, definitions, workflows, + logic**
* A ***good* data model captures how communication + work naturally flow w/in an organization**
* In contrast, a *poor* data model (or *nonexistent* one) is haphazard, confusing, + incoherent
* Some data professionals view data modeling as tedious + reserved for “big enterprises.”
* Data modeling is acknowledged as a good thing to do but is **often ignored in practice**
* **Ideally, every organization should model its data if only to ensure that business logic + rules are translated at the data layer**
* When modeling data, it’s **critical to focus on translating the model to business outcomes**
* **A good data model should correlate w/ impactful business decisions**
* Ex: A “customer” might mean different things to different departments in a company
* Is someone who’s bought from you over the last 30 days a customer? What if they haven’t bought from you in the previous six months or a year?
* Carefully defining + modeling this customer data can have a massive impact on downstream reports on customer behavior or the creation of customer churn models whereby the time since the last purchase is a critical variable
* A ***good* data model contains *consistent* definitions**
* In practice, **definitions are often messy throughout a company**
* Our discussion focuses mainly on **batch data modeling** since that’s **where most data modeling techniques arose,** but we will also look at some approaches to modeling streaming data + some general considerations for modeling

##### Conceptual, Logical, and Physical Data Models

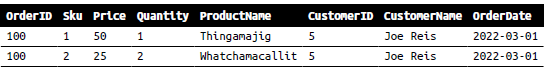
* When modeling data, the **idea is to move from *abstract* modeling concepts to *concrete* implementation**
* Along this continuum, **3 main data models are conceptual, logical, + physical**, which form the basis for the various modeling techniques described later on:
* **Conceptual**
* Contains **business logic + rules and describes the system’s data**, such as **schemas, tables, and fields (names and types)**
* When creating a conceptual model, it’s **often helpful to visualize it in an entity-relationship diagram (ERD)**, a standard tool for visualizing the relationships among various **entities** in your data (orders, customers, products, etc.)
* Ex: An ERD might encode the connections among customer ID, customer name, customer address, + customer orders
* ***Visualizing* entity relationships is *highly* recommended for designing a coherent conceptual data model**
* **Logical**
* **Details how conceptual models will be implemented *in practice* by adding significantly more detail**
* Ex: Would add info on the *types* of customer ID, customer names, + custom addresses
* In addition, we would map out PK’s + FK’s
* **Physical**
* **Defines how the logical model will be implemented in a database system**
* We’d add *specific* databases, schemas, + tables to a logical model, including configuration details
* **Successful data modeling involves business stakeholders at the inception of the process**
* DE’s **need to obtain definitions and business goals for the data**
* Modeling data should be a full-contact sport whose **goal is to provide the business w/ quality data for actionable insights and intelligent automation**
* This is a practice that **everyone must continuously participate in**
* Another important consideration for data modeling is the **grainof the** **data**, which is **the resolution at which data is stored and queried**
* The grain is **typically at the level of a PK** in a table (like customer ID, order ID, or product ID)
* It’s **often accompanied by a date or timestamp for increased fidelity**
* Ex: Suppose a company has just begun to deploy BI reporting
* The company is small enough that the same person is filling the role of DE and analyst
* A request comes in for a report that summarizes daily customer orders
* Specifically, the report should list all customers who ordered, the number of orders they placed that day, and the total amount they spent
* This report is inherently **coarse-grained**, as it **contains no details on spending per order or the items in each order**
* It’s tempting for a DE/analyst to ingest data from the production orders database + boil it down to a reporting table w/ only the basic aggregated data required for the report
* *However*, this would entail starting over when a request comes in for a report w/ finer-grained data aggregation
* Since the DE is actually quite experienced, they elect to create tables w/ detailed data on customer orders, including each order, item, item cost, item IDs, etc.
* Essentially, their tables contain ALL details on customer orders, + the data’s grain is at the customer-order level
* This customer-order data can be analyzed as is, or aggregated for summary statistics on customer order activity
* In general, **strive to model your data at the lowest level of grain possible**.
* From here, it’s **easy to aggregate this highly granular dataset**
* The reverse isn’t true, + it’s **generally impossible to restore details that have been aggregated away**

##### Normalization

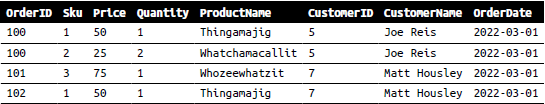
* **Normalization**is **a** **database data modeling practice** that **enforces strict control over the relationships of tables + columns w/in a database**
* The **goal of normalization is to remove the redundancy of data within a database and ensure referential integrity**
* Basically, it’s **Don’t Repeat Yourself (DRY) applied to data in a database**
* For more details on the DRY principle, see *The Pragmatic Programmer* by David Thomas and Andrew Hunt (Addison-Wesley Professional, 2019)
* Normalization is **typically applied to RDBs containing tables w/ rows and columns/fields**
* It was first introduced by RDB pioneer Edgar Codd in the early 1970s, who outlined **four main objectives of normalization**:
* **To free the collection of relations from undesirable insertion, update, + deletion dependencies**
* **To reduce the need for restructuring the collection of relations, as new types of data are introduced, + thus increase the lifespan of application programs**
* **To make the relational model more informative to users**
* **To make the collection of relations neutral to the query statistics, where these statistics are liable to change as time goes by**
* Codd introduced the idea of **normal forms**, which are **sequential**, w/ **each form incorporating the conditions of prior forms**
* The first 3 normal forms are:
* **Denormalized:** No normalization, nested + redundant data is allowed
* **First normal form (1NF):** Each column is unique + has a single value, + the table has a unique PK
* **Second normal form (2NF):** The requirements of 1NF, + partial dependencies are removed
* **Third normal form (3NF):** The requirements of 2NF, + each table contains only relevant fields related to its PK + has no transitive dependencies
* A **unique primary keyis a single field or set of multiple fields that uniquely determines rows in the table**
* **Each key value occurs at most once, otherwise a value would map to multiple rows in the table**
* Thus, **every other value in a row is dependent on (“can be determined from”) the key**
* A **partial dependency****occurs when a subset of fields in a composite key can be used to determine a non-key column of the table**
* A **transitive dependency****occurs when a non-key field depends on another non-key field**
* Let’s look at stages of normalization (from denormalized to 3NF) using an ecommerce example of customer orders



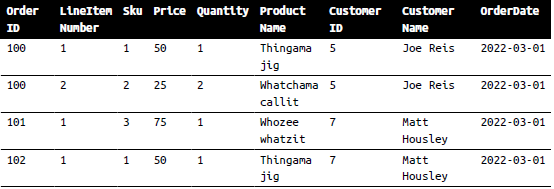
* First, this denormalized *OrderDetail* table contains 5 fields, + the PK is *OrderID*
* Notice that the *OrderItems* field contains a nested object w/ 2 SKUs along w/ their price, quantity, + name
* *To convert this data to* ***1NF***, move *OrderItems* into 4 fields



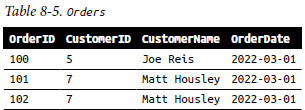
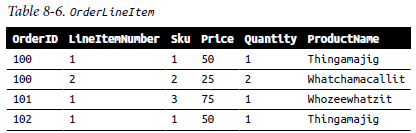
* Now we have an *OrderDetail* table in which fields do NOT contain repeats or nested data
* The **problem** is that **now we don’t have a unique PK**
* That is, “*100” occurs in the OrderID column in 2 different rows*
* To get a better grasp of the situation, look at a larger sample from our table



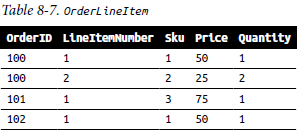
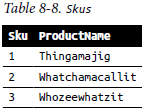
* To create a unique primary (*composite*) key, number the lines in each order by adding a column called *LineItemNumber*



* The **composite key** (*OrderID*, *LineItemNumber*) is now a ***unique* PK**
* To reach **2NF**, we need to **ensure that no partial dependencies exist**
* Again, a **partial dependency** is a **non-key column that is fully determined by a subset of the columns in the unique primary (composite) key**
* ***Partial dependencies can occur only when the PK is composite***
* In our case, the last 3 columns are determined by order number
* To fix this problem, split *OrderDetail* into 2 tables: *Orders* and *OrderLineItem*

* The composite key (*OrderID, LineItemNumber*) is a unique PK for *OrderLineItem*, while *OrderID* is a PK for *Orders*
* To reach **3NF**, notice that *Sku* determines *ProductName* in *OrderLineItem*
* That is, *Sku* *depends* on the *composite* key, and *ProductName* depends on *Sku*, which is a **transitive dependency**
* Let’s break *OrderLineItem* into *OrderLineItem* and *Skus*

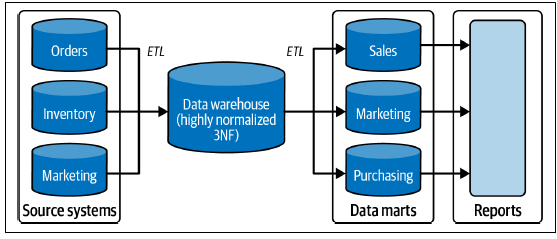
* Now, both *OrderLineItem* and *Skus* are in 3NF
* But notice that *Orders* does *not* satisfy 3NF
* What transitive dependencies are present? How would you fix this?
* **Additional normal forms exist** (up to 6NF in the Boyce-Codd system), but these are **much less common than the first three**
* **A database is usually considered “normalized” if it’s in 3NF**
* The **degree of normalization that you should apply to your data depends on your use case**
* No one-size-fits-all solution exists, especially in databases where some denormalization presents performance advantages
* Although **denormalization may seem like an** **antipattern**, it’s **common in many OLAP systems that store semi-structured data**
* **Study normalization conventions and database best practices to choose an appropriate strategy**

##### Techniques for Modeling Batch Analytical Data

* **When describing data modeling for data lakes or DW’s**, you should **assume** **the raw data takes many forms (e.g., structured and semistructured),** but the ***output* is a *structured* data model of rows and columns**
* However, ***several* approaches to data modeling can be used in these environments**
* The big approaches you’ll likely encounter are **Kimball**, **Inmon**, and **Data Vault**.
* **In practice, some of these techniques can be combined**
* Ex: Can see some data teams start w/ Data Vault + then add a Kimball star schema alongside it
* We’ll also look at **wide** and **denormalized data models** + other batch data-modeling techniques you should have in your arsenal
* NOTE: Coverage of the first 3 approaches (Inmon, Kimball, Data Vault) is cursory + hardly does justice to their respective complexity + nuance
* At the end of each section, note the canonical books from their creators
* **For a DE, these books are must-reads, if only to understand how and why data modeling is central to batch analytical data**

###### Inmon

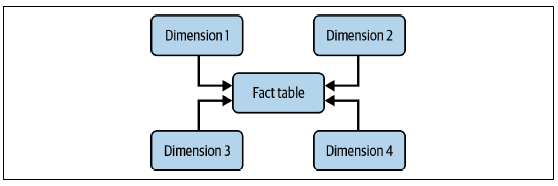
* The father of the DW, Bill Inmon, created his approach to data modeling in 1989
* Before the DW, analysis would often occur *directly* on the source system itself, w/ the obvious consequence of bogging down production transactional databases w/ long-running queries
* **The goal of the DW was to separate the source system from the analytical system**
* Inmon defines a DW the following way (*Building the Data Warehouse* (Hoboken: Wiley, 2005)):
* *A DW is a* ***subject-oriented, integrated, nonvolatile, + time-variant collection of data in support of management’s decisions***
* *The DW* ***contains granular corporate data***
* *Data in the DW is* ***able to be used for many different purposes, including sitting + waiting for future requirements which are unknown today***
* The **4 critical parts of a DW** can be described as follows:
* **1) Subject-oriented**: The **DW** **focuses on a *specific* subject area**, such as sales or marketing
* First, the ***logical* model must focus on a *specific* area**
* For instance, if the subject orientationis “sales,” then the logical model contains all details related to sales (business keys, relationships, attributes, etc.)
* **2) Integrated**: **Data** from **disparate sources is consolidated + normalized**
* Details from a logical model are integratedinto a consolidated + highly-normalized data model
* **3) Nonvolatile**: **Data** **remains unchanged after being stored in a DW**
* **4) Time-variant**: **Varying time ranges can be queried**
* i.e., Storing data in **both a nonvolatile and time-variant**way, **meaning you can (theoretically) query the original data for as long as storage history allows**
* The **Inmon DW *must* strictly adhere to *all four* of these critical parts *in support of management’s decisions***
* This is a subtle point, but it **positions the DW for *analytics*, + *NOT* for OLTP**
* **Another key characteristic of Inmon’s DW** (*Building the Data Warehouse* (Hoboken: Wiley, 2005)):
* *The 2nd salient characteristic of the data warehouse is that it is* ***integrated***
* ***Of all the aspects of a DW, integration is the most important***
* *Data is fed* ***from multiple, disparate sources*** *into the DW*
* ***As the data is fed, it is converted, reformatted, re-sequenced, summarized, etc.***
* *The* ***result*** *is that data (once it resides in the DW) has a* ***single physical corporate image***
* W/ **Inmon’s DW, data is integrated from *across* the organization in a *granular*, highly-normalized ER model, w/ a relentless emphasis on *ETL***
* B/c of the **subject-oriented nature** of the DW, the **Inmon DW consists of key source databases and information systems used in an organization**
* **Data from key business source systems is ingested + integrated into a highly normalized (3NF) DW *that often closely resembles the normalization structure of the source system itself***
* **Data is brought in incrementally, starting w/ the highest-priority business areas**
* The **strict normalization requirement ensures as little data duplication as possible, which leads to fewer downstream analytical errors b/c data won’t diverge or suffer from redundancies**
* The **DW represents a “single source of truth,” which supports the overall business’s information requirements**
* The **data is presented for downstream reports + analysis via business and department-specific data marts, which may *also* be denormalized**
* Let’s look at how an Inmon DW is used for ecommerce



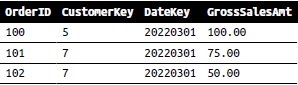
* The **business source systems** are orders, inventory, + marketing
* The data from these source systems are **ETL-ed to the DW + stored in 3NF**
* ***Ideally*, the DW *holistically* encompasses the business’s information**
* **To serve data for *department-specific* information requests, ETL processes take data from the DW, *transform* it, + place it in downstream data marts to be viewed in reports**
* A **popular option for modeling data in a data mart is a** **star schema** (see Kimball), though ***any* data model that provides easily accessible information is also suitable**
* Above, **the departments** (sales, marketing, + purchasing) **have their *own* star schema**, fed upstream from the granular data in the DW
* This **allows each department to have its *own* data structure that’s unique + optimized to *its specific needs***
* W. H. (Bill) Inmon continues to innovate in the DW space, currently focusing on **textual ETL** in the DW (process that takes text, integrates text, + produces the text in a form compatible w/ the analytical processes that already exist in a corporation)
* <https://www.techtarget.com/searchdatamanagement/feature/Bill-Inmons-data-warehouse-approach-tackles-text-analysis>
* *The Textual Warehouse* by [Bill Inmon](https://shop.harvard.com/search/author/%22Inmon%2C%20Bill%22), [Ranjeet Srivastava](https://shop.harvard.com/search/author/%22Srivastava%2C%20Ranjeet%22) (Technics, 2021)
* <https://shop.harvard.com/book/9781634629546>
* He’s also a prolific writer + thinker, writing over 60 books + countless articles
* For further reading about Inmon’s DW, please refer to:
* Zentut’s “Bill Inmon Data Warehouse” web page
* <https://www.zentut.com/data-warehouse/bill-inmon-data-warehouse/>
* *“The Evolution of the Corporate Information Factory”*
* <https://www.ewsolutions.com/evolution-corporate-information-factory/>
* <https://shop.harvard.com/search/author/%22Inmon%2C%20Bill%22>
* *Building the Data Warehouse* (Wiley, 4e, 2005)
* <https://www.wiley.com/en-us/Building+the+Data+Warehouse,+4th+Edition-p-9780764599446>
* *Corporate Information Factory* (Wiley, 2e, 2002)
* <https://www.wiley.com/en-us/Corporate+Information+Factory%2C+2nd+Edition-p-9780471437505>
* *Building the Data Lakehouse* (Technics, 2021)
* <https://shop.harvard.com/book/9781634629669>
* *The Unified Star Schema* (Technics, 2020)
* <https://shop.harvard.com/book/9781634628877>

###### Kimball

* If there are spectrums to data modeling, **Kimball is very much on the opposite end of Inmon**
* Created by Ralph Kimball in the early 1990s, this approach to data modeling **focuses less on normalization, and in some cases accepting denormalization**
* As *Inmon* says about the difference between the DW and data mart, “*A data mart is never a substitute for a DW*.” (*Building the Data Warehouse* (Hoboken: Wiley, 2005)):
* The **Inmon model is top-down, as it integrates data from across the business in the DW, and serves department-specific analytics via data marts**
* The **Kimball model is bottom-up, encouraging you to model + serve department or business analytics *in the DW itself***
* ***Inmon argues this approach skews the definition of a DW***
* The **Kimball approach effectively makes the data mart the DW itself**
* This **may enable faster iteration + modeling than Inmon**, with the **trade-off of potential looser data integration, data redundancy, + duplication**
* In Kimball’s approach, **data is modeled with 2 general types of tables: facts and dimensions**
* Think of a **fact table**as **a table of numbers**, and **dimension tables**as ***qualitative* data *referencing* a fact**
* Dimension tables surround a *single* fact table in a relationship called a **star schema**



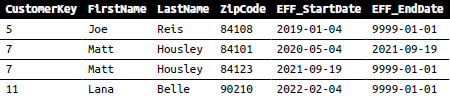
* NOTE: Although dimensions + facts are often associated w/ Kimball, they were first used at General Mills + at Dartmouth University in the 1960s, + had early adoption at Nielsen and IRI, among other companies
* **Fact tables**
* The first type of table in a star schema is the fact table, which **contains *factual*, quantitative, and event-related data**
* **Data in a fact table is immutable because facts relate to *events***
* Therefore, **fact tables don’t change and are append-only**
* Fact tables are **typically narrow and long**, meaning they have **not a lot of columns but a lot of rows that represent events**
* **Fact tables should be at the lowest grain possible**
* **Queries against a star schema *start* w/ the fact table**
* **Each row of a fact table should represent the grain of the data**
* **Avoid aggregating or deriving data w/in a fact table.**
* **If you *need* to perform aggregations or derivations, do so in a *downstream* query, data mart table, or view**
* Finally, **fact tables don’t reference other fact tables**, as they **reference only dimensions**
* Let’s look at an example of an elementary fact table
* A common question in a company might be, “Show me gross sales, by each customer order, by date”



* **Again, facts should be at the lowest grain possible**
* In this case, the ***orderID*** of the sale, customer, date, + gross sale amount
* **Notice that the data types in the fact table are all numbers (integers and floats)**, + there are *no* strings
* Also, in this example, *CustomerKey* = 7 has *two* orders on the *same* day, **reflecting the grain of the table**
* Instead, **the fact table has keys that reference dimension tables containing their respective attributes**, such as the customer + date information
* The **gross sales amount represents the total sale for the sales *event***
* **Dimension tables**
* The 2nd primary type of table in a Kimball data model = **Dimension tables**, which **provide the reference data, attributes, + relational context for the events stored in fact tables**
* Dimension tables are **smaller than fact tables + take an opposite shape, typically wide and short (many columns and fewer rows)**
* **When joined to a fact table, dimensions can describe the events’ what, where, and when**
* **Dimensions are DE-normalized, w/ the possibility of duplicate data**
* This is **OK in the Kimball data model**
* Let’s look at the 2 dimensions referenced in the earlier fact table example, date and customer



* **In a Kimball data model, dates are typically stored in a date dimension, allowing you to reference the date key (DateKey) between the fact + date dimension table**
* W/ the date dimension table, you can easily answer questions like, “What are my total sales in the first quarter of 2022?” or “How many more customers shop on Tuesday than Wednesday?”
* Notice we have 5 fields in addition to the date key
* The **beauty of a date dimension is that you can add as many new fields as makes sense to analyze your data**



* The customer dimension contains several fields that describe the customer: first and last name, zip code, and a couple of peculiar-looking date fields.
* These date fields illustrate another concept in the Kimball data model: a Type 2 **slowly changing dimension** (described in greater detail next)
* Ex: Look at *CustomerKey* = 5, w/ the EFF\_*StartDate* (effective start date) of 2019-01-04 and an *EFF\_EndDate* of 9999-01-01
* This means Joe Reis’s customer record was created in the customer dimension table on 2019-01-04 and has an end date of 9999-01-01
* *What does this end date mean?* 🡪 It means the customer record is **active and isn’t changed**
* Now look at Matt Housley’s customer record (*CustomerKey* = 7)
* Notice the 2 entries for Housley’s start date: 2020-05-04 and 2021-09-19
* It looks like Housley changed his zip code on 2021-09-19, resulting in a change to his customer record
* **When the data is queried for the most recent customer records, you will query where the end date is equal to 9999-01-01**
* A **slowly changing dimension (SCD)** **is necessary to track changes in dimensions**
* The preceding example is a **Type 2 SCD: a new record is inserted when an existing record changes**
* Though SCDs can go up to 7 levels, let’s look at the 3 most common ones:
* **Type 1 SCD: Overwrite existing dimension records**
* This is super simple and means you have **no access to the deleted historical dimension records**
* **Type 2 SCD: Keep a full history of dimension records via a new record/row**
* When a **record changes**, that **specific record is flagged as changed**, and a **new** **dimension record (row) is created that reflects the current status of the attributes**
* Ex: Housley moved to a new zipcode, which triggered his initial record to reflect an effective end date, + a newrecord was created to show his new zip code
* **Type 3 SCD: Keep a full history of dimension records via a new column/field**
* Similar to a Type 2 SCD, but instead of creating a new row, **a change in a Type 3 SCD creates a new *column/field***
* Using the above example, let’s see what this looks like as a Type 3 SCD in the following tables



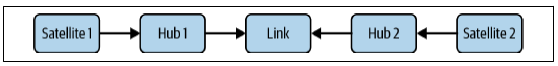
* Here, Housley lives in the 84101 zip code
* When Housley moves to a new zip code, the **Type 3 SCD creates 2 new fields**, one for his new zip code and the date of the change



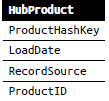
* The original zip code field is also renamed to reflect that this is the older record
* Of the types of SCDs described, **Type 1 is the default behavior of most DW’s, + Type 2 is the one most commonly seen used in practice**
* There’s a *lot* to know about dimensions, this section is just a starting point to get familiar w/ how dimensions work + how they’re used
* **Star schema**
* Now that you have a basic understanding of facts and dimensions, it’s time to integrate them into a **star schema, which represents the data model of the business**
* ***Unlike highly-normalized approaches to data modeling*, the star schema is a fact table surrounded by the necessary dimensions**
* This **results in fewer joins than other data models, which speeds up query performance**
* Another advantage is it’s **arguably easier for business users to understand + use**
* Note that the star schema **shouldn’t reflect a particular report**, though you ***can* model a report in a downstream data mart or directly in a BI tool**
* The star schema **should capture the facts + attributes of your *business logic* + also be flexible enough to answer the respective critical questions**
* **Because a star schema has** ***one* fact table, sometimes you’ll have multiple star schemas that address different facts of the business**
* You should **strive to reduce the number of dimensions whenever possible since this reference data can potentially be reused among different fact tables**
* A **dimension that’s reused across multiple star schemas, thus sharing the same fields, is called a conformed dimension**
* A conformed dimension **allows you to combine multiple fact tables across multiple star schemas**
* **Remember, redundant data is OK with the Kimball method, but avoid replicating the same dimension tables to avoid drifting business definitions and data integrity**
* The **Kimball data model + star schema have a lot of nuance**
* **Be aware that the Kimball model is appropriate only for *batch* data, + NOT for streaming data**
* Because the Kimball data model is popular, there’s a good chance you’ll run into it

###### Data Vault

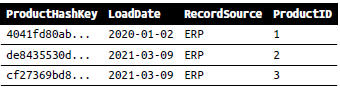
* Whereas Kimball and Inmon focus on the structure of business logic in the data warehouse, the **Data Vault** offers a different approach to data modeling.11
* *NOTE: The Data Vault has 2 versions, 1.0 and 2.0, we focus on Data Vault 2.0, but will call it Data Vault for the sake of brevity*
* Created in the 1990s by Dan Linstedt, the **Data Vault methodology separates the structural aspects of a source system’s data from its attributes**
* Instead of representing business logic in facts, dimensions, or highly normalized tables, a **Data Vault simply loads data from source systems directly into a handful of purpose-built tables in an insert-only manner**
* Unlike the other data modeling approaches above, there’s **no notion of good, bad, or conformed data in a Data Vault**
* Data moves fast these days, and **data models need to be agile, flexible, + scalable**, so **the Data Vault methodology aims to meet this need**
* The **goal of this methodology is to keep the data as closely aligned to the business as possible, even while the business’s data evolves**
* A **Data Vault model consists of 3 main types of tables: hubs, links, and satellites**



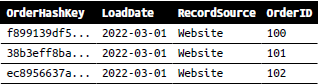
* In short, a **hub stores business keys**, a **link** **maintains relationships among business keys**, and a **satelliterepresents a business key’s attributes + context**
* A user will **query a hub**, which will **link to a satellite table** containing the query’s relevant attributes
* **Hubs**
* Q**ueries often involve searching by a business key,** such as a customer ID or an order ID
* A **hub is the central entity of a Data Vault that retains a record of all unique business keys loaded into the Data Vault**
* A hub **always contains the following standard fields**:
* **Hash key:** The PK used to join data between systems
* This is a calculated hash field (MD5 or similar)
* **Load date:** The date the data was loaded into the hub.
* **Record source:** The source from which the unique record was obtained
* **Business key(s):** The key used to identify a unique record
* It’s important to note that **a hub is insert-only**, and **data is not altered in a hub**
* Once data is loaded into a hub, it’s permanent
* **When designing a hub, identifying the business key is critical**
* **Ask yourself: *What is the identifiable business element?***
* Kent Graziano, “Data Vault 2.0 Modeling Basics,” Vertabelo, October 20, 2015
* <https://oreil.ly/iuW1U>
* **Put another way, *how do users commonly look for data*?**
* **Ideally, this is discovered as you build the conceptual data model of your organization and before you start building your Data Vault**
* Using our ecommerce scenario, let’s look at an example of a hub for products
* First, let’s look at the **physical design of a product hub**



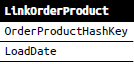
* In practice, the product hub looks like this when populated with data



* In this example, 3 different products are loaded into a hub from an ERP system on 2 separate dates
* While we’re at it, let’s create another hub for *orders* using the same schema as *HubProduct*, and populate it with some sample order data

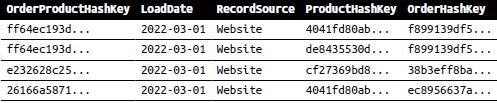


* **Links**
* A link table**tracks the relationships of business keys between hubs**
* Link tables **connect (various) hubs, ideally at the lowest possible grain**
* B/c link tables connect data from various hubs, they are **many to many**
* The **Data Vault model’s relationships are straightforward + handled through changes to the links**
* This **provides excellent flexibility in the inevitable event that the underlying data changes**
* You **simply create a *new* link that ties business concepts (or hubs) to represent the new relationship**
* Now let’s look at ways to view data contextually using satellites.
* Back to our ecommerce example, we’d like to **associate orders with products**
* Let’s see what a link table might look like for orders and products

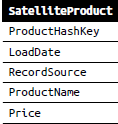




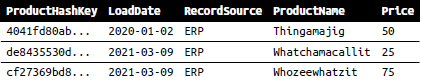
* When the *LinkOrderProduct* table is populated, here’s what it looks like



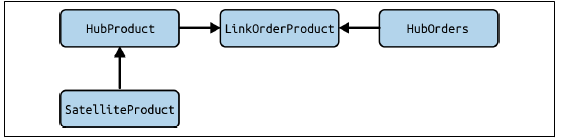
* Note that we’re using the order’s record source in this example
* **Satellites**
* We’ve described relationships between hubs and links that involve keys, load dates, + record sources
* How do you get a sense of what these relationships *mean*?
* **Satellites**are **descriptive attributes that give meaning and context to hubs (business keys)**
* Satellites **can connect to either hubs or links**
* The **only *required* fields in a satellite are a PK consisting of the business key of the parent hub + a load date**.
* **Beyond that**, a satellite **can contain however many attributes that make sense**
* Example of a satellite for the Product hub



* In this example, the *SatelliteProduct* table contains additional information about the product, such as product name and price
* And here’s it is w/ some sample data



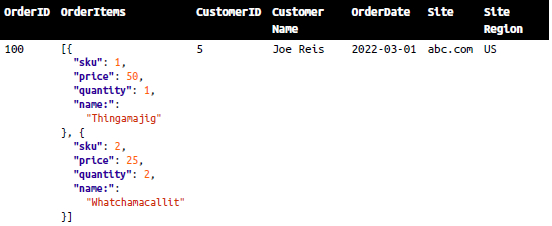
* Let’s **tie this all together and join the hub, product, and link tables into a Data Vault**



* Other types of Data Vault tables exist, including point-in-time (PIT) and bridge tables
* We don’t cover these here, but mention them because the Data Vault is quite comprehensive
* This has just been an overview of the Data Vault’s power.
* Unlike other data modeling techniques discussed, **in a Data Vault, the business logic is created and interpreted *when the data from these tables is queried***
* Please be aware that **the Data Vault model can be used with other data modeling techniques**
* It’s n**ot unusual for a Data Vault to be the landing zone for analytical data, after which it’s separately modeled in a DW, commonly using a star schema**
* The Data Vault model also can be adapted for NoSQL and streaming data sources

###### Wide Denormalized Tables

* The **strict modeling** approaches we’ve described, especially Kimball and Inmon, were **developed when DW’s were expensive, on premises, + heavily resource-constrained w/ tightly coupled compute and storage**
* While batch data modeling has traditionally been associated w/ these strict approaches, **more relaxed approaches are becoming more common**
* There are reasons for this
* **1) The popularity of the cloud means that storage is dirt cheap**
* It’s cheaper to store data than agonize over the optimum way to represent the data in storage
* **2) The popularity of nested data (JSON and similar) means schemas are flexible in source and analytical systems**
* You have the option to rigidly model your data as described, or you can choose to throw all of your data into a single **wide table**
* A **wide table** is just what it sounds like: a **highly DE-normalized and very wide collection of many fields, typically created in a *columnar* database**
* A **field may be a single value *OR* contain nested data**
* The **data is organized along w/ one or multiple keys, which are closely tied to the grainof the data**
* A **wide table can potentially have thousands of columns, whereas < 100 are typical in RDBs**
* Wide tables are **usually sparse** (the vast majority of entries in a given field may be null)
* This is **extremely expensive in a traditional RDB b/c the database allocates a fixed amount of space for each field entry**
* **Nulls take up virtually no space in a columnar database**
* A **wide schema in an RDB dramatically slows reading b/c each row must allocate *all* the space specified by the wide schema, + the RDB must read the contents of *each row* *in its entirety***
* On the other hand, **a columnar database reads only columns selected in a query, + reading nulls is essentially free**
* **Wide tables generally arise through schema evolution, as engineers gradually add fields over time.**
* **Schema evolution in an RDB is a slow and resource-heavy process**
* **In a columnar database, adding a field is initially just a change to metadata**
* As data is written into the new field, new files are added to the column
* **Analytics queries on wide tables often run faster than equivalent queries on highly normalized data requiring many joins**
* **Removing joins can have a huge impact on scan performance**
* The **wide table simply contains all of the data you would’ve joined in a more rigorous modeling approach**
* **Facts and dimensions are represented in the same table**
* The lack of data model rigor also means not a lot of thought is involved: **Load your data into a wide table and start querying it**
* Especially w/ schemas in source systems becoming more adaptive and flexible, **this data usually results from high-volume transactions, meaning there’s a lot of data**
* **Storing this as nested data in your analytical storage has a lot of benefits**
* **Throwing all of your data into a single table** might seem like heresy for a hardcore data modeler, and there’s been plenty of **criticism**
* The biggest criticism is **as you blend your data, you lose the business logic in your analytics**
* Another downside is the **performance of updates to things like an element in an array**, which can be very painful
* Here’s example of a wide table, using the original denormalized table from our earlier normalization example



* This table can have many more columns (hundreds or more!)
* We include only a handful of columns for brevity + ease of understanding
* As you can see, **this table combines various data types, represented along a grain of orders for a customer on a date**
* **Use a wide table when you don’t care about data modeling, or when you have a lot of data that needs more flexibility than traditional data-modeling rigor provides**
* **Wide tables also lend themselves to streaming data**
* As data moves toward fast-moving schemas and streaming-first, **expect to see a new wave of data modeling, perhaps something along the lines of “relaxed normalization”**

What If You Don’t Model Your Data?

* You also have the option of ***NOT* modeling your data**
* In this case, just **query data sources directly**
* This pattern is **often used**, especially when **companies are just getting started + want to get quick insights or share analytics with their users**
* While it allows you to get answers to various questions, you should **consider the following**:
* **If I don’t model my data, how do I know the results of my queries are consistent?**
* **Do I have proper definitions of business logic in the source system, + will my query produce truthful answers?**
* **What query load am I putting on my source systems, + how does this impact users of these systems?**
* At some point, you’ll probably gravitate toward a stricter batch data model paradigm + a dedicated data architecture that doesn’t rely on the source systems for the heavy lifting

##### Modeling Streaming Data

* **Whereas many data-modeling techniques are well established for batch, this is not the case for streaming data**
* Because of the **unbounded + continuous nature** of streaming data, **translating batch techniques** like Kimball to a streaming paradigm **is tricky, if not impossible**
* Ex: Given a stream of data, how would you continuously update a Type-2 SCD w/out bringing your DW to its knees?
* **The world is evolving from batch to streaming and from on premises to the cloud**
* The **constraints of the older batch methods no longer apply**
* That said, big questions remain about how to model data to balance the need for business logic against fluid schema changes, fast-moving data, + self-service
* *What is the streaming equivalent of the preceding batch data model approaches?*
* **There isn’t (yet) a consensus approach on streaming data modeling**
* Many experts in streaming data systems say that traditional batch-oriented data modeling doesn’t apply to streaming
* Though a few may suggest the Data Vault as an option for streaming data modeling.
* Recall that **2 two main types of streams exist: event streams and CDC**
* ***Most of the time*, the shape of the data in these streams is semi-structured**, such as JSON
* **The challenge w/ modeling streaming data is that the payload’s schema might change on a whim**
* Ex: You have an IoT device that recently upgraded its firmware + introduced a new field
* In that case, it’s possible that your downstream destination DW or processing pipeline isn’t aware of this change + thus breaks
* Ex: A CDC system might recast a field as a different type (say, a string instead of an International Organization for Standardization (ISO) datetime format)
* Again, **how does the destination handle this seemingly random change?**
* **Streaming data experts overwhelmingly suggest you anticipate changes in the source data and keep a flexible schema**
* This means there’s **no rigid data model in the analytical database**
* **Instead, assume the source systems are providing the correct data with the right business definition and logic, as it exists today**
* And **b/c storage is cheap, store the recent streaming + saved historical data in a way they can be queried together**
* **Optimize for comprehensive analytics against a dataset w/ a flexible schema**
* Furthermore, **instead of reacting to reports, why not create automation that responds to anomalies + changes in the streaming data instead?**
* **The world of data modeling is changing, + a sea change may soon occur in data model paradigms**
* New approaches will likely incorporate metrics + semantic layers, data pipelines, + traditional analytics workflows in a streaming layer that sits directly on top of the source system
* Since data is being generated in real time, the notion of artificially separating source and analytics systems into 2 distinct buckets may not make as much sense as when data moved more slowly + predictably
* Time will tell, + see more on the future of streaming data in Chapter 11

#### Transformations

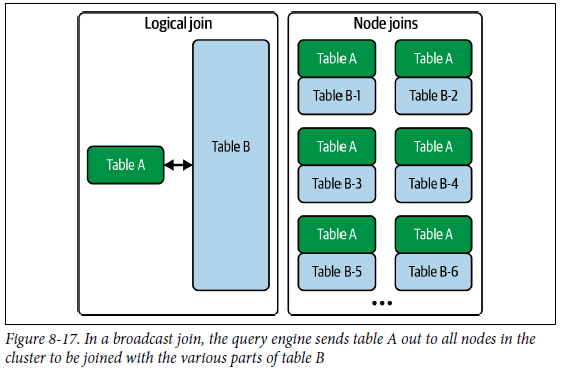
* Inmon: “*The net result of* ***transforming*** *data is the* ***ability to unify and integrate data****.* ***Once*** *data is* ***transformed****, the* ***data can be viewed as a single entity****. But* ***without transforming data, you cannot have a unified view of data across the organization****”*
* Bill Inmon, “Avoiding the Horrible Task of Integrating Data,” LinkedIn Pulse, March 24, 2022
* <https://oreil.ly/yLb71>
* Now that we’ve covered queries and data modeling, you might be wondering, if I can model data, query it, + get results, why do I need to think about transformations?
* **Transformations manipulate, enhance, + save data for downstream use, increasing its value in a scalable, reliable, + cost-effective manner**
* Imagine running a query every time you want to view results from a particular dataset
* You’d run the same query dozens or hundreds of times a day
* Imagine that this query involves parsing, cleansing, joining, union-ing, + aggregating across 20 datasets
* To further exacerbate the pain, the query takes 30 minutes to run, consumes significant resources, and incurs substantial cloud charges over several repetitions.
* You + your stakeholders would probably go insane
* Thankfully, you **can *save the results of your query* instead, or at least run the most compute-intensive portions only once, so subsequent queries are simplified**
* A **transformation differs from a query:**
* **1) Persistence**
* A **queryretrieves the data from various sources based on filtering and join logic**
* A **transformation *persists* results for consumption by additional transformations or queries**
* These **results may be stored ephemerally or permanently**
* **2) Complexity**
* You’ll **likely build complex pipelines that combine data from multiple sources + reuse intermediate results for multiple final outputs**
* These complex **pipelines might normalize, model, aggregate, or featurize data**
* While you *can* build complex dataflows in single queries using CTE’s, scripts, or DAGs, this quickly becomes unwieldy, inconsistent, + intractable
* Enter transformations
* **Transformations critically rely on one of the major undercurrents in this book: orchestration**.
* **Orchestration combines many discrete operations, such as intermediate transformations, that store data temporarily or permanently for consumption by downstream transformations or serving**
* **Increasingly, transformation pipelines span not only multiple tables + datasets but also multiple systems**

##### Batch Transformations

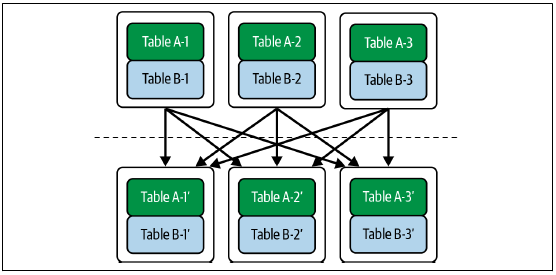
* **Batch transformationsrun on discrete chunks of data, in contrast to streaming transformations, where data is processed continuously as it arrives**
* Batch transformations **can run on a fixed schedule** (e.g., daily, hourly, or every 15 minutes) to support ongoing reporting, analytics, + ML models
* There are various batch transformation patterns and technologies.

###### a) Distributed Joins

* The basic idea behind **distributed joins** = we **need to break a logical join(the join defined by the query logic) into much smaller node joinsthat run on individual servers in the cluster**
* The basic distributed join patterns apply whether one is in MapReduce, BigQuery, Snowflake, or Spark, though the details of intermediate storage between processing steps vary (on disk or in memory)
* In the **best-case scenario**, the **data on one side of the join is small enough to fit on a *single* node (broadcast join)**
* **But often, a more resource-intensive shuffle hash joinis required.**
* A **broadcast join**is **generally asymmetric**, with **one large table distributed across nodes and one small table that can easily fit on a single node**



* The **query engine “broadcasts” the small table** **(table A) out to all nodes, where it gets joined to the parts of the large table (table B)**
* Broadcast joins are **far less compute intensive than shuffle hash joins**
* In practice, table A is often a **down-filtered** larger table that the query engine collects and broadcasts
* **One of the top priorities in query optimizers is join reordering**
* W/ the **early application of filters, + movement of small tables to the left (for left joins),** it is **often possible to dramatically reduce the amount of data that is processed in each join**
* **Prefiltering data to create broadcast joins where possible can dramatically improve performance + reduce resource consumption**
* If **neither table is small enough to fit on a single node**, the query engine will use a **shuffle hash join**



* The same nodes are represented above + below the dotted line
* The area above the dotted line represents the initial partitioning of tables A and B across the nodes
* **In general, this partitioning will have no relation to the join key**
* A **hashing scheme is used to repartition data by join key**
* In this example, the hashing scheme will partition the join key into 3 parts, w/ each part assigned to a node
* The data is then reshuffled to the appropriate node, + the new partitions for tables A and B on each node are joined
* **Shuffle hash joins are generally more resource intensive than broadcast joins**

###### b) ETL, ELT, and Data Pipelines

* **Reminder: A widespread transformation pattern dating to the early days of RDBs is a batch ETL**
* ***Traditional* ETL relies on an *external* transformation system to pull, transform, + clean data while preparing it for a target *schema*, such as a data mart or a Kimball star schema**
* The **transformed data would then be loaded into a target *system***, such as a DW, where business **analytics could be performed**
* The **ETL pattern itself was driven by the limitations of both source + target systems**
* The extract phase tended to be a major bottleneck, w/ the constraints of the **source RDBMS limiting the rate at which data could be pulled**
* And the transformation was handled in a dedicated system b/c the **target system was extremely resource-constrained in both storage *and* CPU capacity**
* A **now-popular evolution of ETL is ELT**
* As DW systems have grown in performance + storage capacity, it has become **common to simply extract raw data from a source system, import it into a DW w/ minimal transformation, + then clean + transform it directly in the DW system** (See Chapter 3)
* A second, slightly different notion of ELT was popularized w/ the emergence of **data lakes:**
* In this version, the **data is *NOT* transformed at the time it’s loaded**
* Indeed, **massive quantities of data may be loaded w/ no preparation + no plan whatsoever**
* The **assumption is that the transformation step will happen at some *undetermined* future time**
* **Ingesting data without a plan is a great recipe for a data swamp**
* Inmon: *I’ve always been a fan of ETL because of the fact that ETL forces you to transform data before you put it into a form where you can work with it*
* *But some organizations want to simply take the data, put it into a database, then do the transformation....*
* *I’ve seen too many cases where the organization says, oh we’ll just put the data in and transform it later*
* *And guess what? Six months later, that data [has] never been touched*
* We have also seen that the **line between ETL and ELT can become somewhat blurry in a data lakehouse environment**
* W/ **object storage as a base layer**, it’s **no longer clear what’s in the database + out of the database**
* The **ambiguity is further exacerbated with the emergence of data federation, virtualization, and live tables**. (discussed later)
* **The terms ETLand ELTmaybeshould be applied only at the *micro* level (w/in *individual* transformation pipelines) rather than at the macro level (to describe a transformation pattern for a *whole* organization)**
* Organizations no longer need to standardize on ETL or ELT but can instead **focus on applying the proper technique on a *case-by-case basis* as they build data pipelines**

###### c) SQL and Code-Based Transformation Tools

* At this juncture, the distinction between SQL-based and non-SQL-based transformation systems feels somewhat synthetic
* Since the introduction of Hive on the Hadoop platform, SQL has become a first-class citizen in the big data ecosystem
* Ex: Spark SQL was an early feature of Apache Spark, + streaming-first frameworks such as Kafka, Flink, and Beam also support SQL, with varying features and functionality
* **It is more appropriate to think about SQL-only tools vs. those that support more powerful, general-purpose programming paradigms**
* SQL-only transformation tools span a wide variety of proprietary and OSS options
* **SQL is declarative..., but it can still build complex data workflows**
* We often hear **SQL** dismissed because it is “**not procedural**,” which is technically correct
* **SQL is a declarative language: instead of coding a data processing procedure, SQL writers stipulate the characteristics of their final data in set-theoretic language**, + the **SQL compiler and optimizer determine the steps required to put data in this state**
* People sometimes imply that because SQL is not procedural, it cannot build out complex pipelines, which is false
* **SQL can effectively be used to build complex DAGs using CTE’s, SQL scripts, or an orchestration tool**
* To be clear, **SQL has limits**, but it’s common to see DE’s doing things in Python and Spark that could be more easily and efficiently done in SQL
* For a better idea of such trade-offs, let’s look at a couple of examples of Spark and SQL
* **Example: When to avoid SQL for batch transformations in Spark**
* When determining whether to use native Spark or PySpark code instead of Spark SQL or another SQL engine, ask yourself the following questions:
* *1. How* ***difficult*** *is it* ***to code*** *the transformation* ***in SQL****?*
* Many transformations coded in Spark could be realized in fairly simple SQL statements
* On the other hand, if the transformation is not realizable in SQL, or if it would be extremely awkward to implement, native Spark is a better option
* Ex: We might be able to implement word stemming in SQL by placing word suffixes in a table, joining w/ that table, using a parsing function to find suffixes in words, + then reducing the word to its stem by using a substring function
* However, this sounds like an extremely complex process w/ numerous edge cases to consider
* A more powerful procedural programming language is a better fit here
* *2. How* ***readable and maintainable*** *will the resulting SQL code be?*
* The above word-stemming query example will be neither readable nor maintainable
* *3. Should some of the transformation code be pushed into a custom library for* ***future reuse*** *across the organization?*
* One of the major limitations of **SQL** is that **it *doesn’t* include a natural notion of libraries or reusable code**
* One **exception** is that **some SQL engines** allow you to maintain **user-defined functions (UDFs)** as **objects inside a database**.15
* *Use UDFs responsibly*
* SQL UDFs often perform reasonably well, but JavaScript UDFs can increase query time from a few minutes to several hours
* However, these UDFs *aren’t* committed to a Git repo w/out an external CI/CD system to manage deployment
* Furthermore, **SQL doesn’t have a good notion of reusability for more complex query components**
* Of course, reusable libraries are easy to create in Spark and PySpark
* We will add that **it *IS* possible to recycle SQL in 2 ways**
* 1) Can easily **reuse the *results* of a SQL query by committing to a table or creating a view**
* Often best handled in an orchestration tool such as Airflow so that downstream queries can start once the source query has finished
* 2) **dbt facilitates the reuse of SQL statements + offers a templating language that makes customization easier**
* **Example: Optimizing Spark and other processing frameworks**
* Spark acolytes often complain that **SQL doesn’t give them control over data processing**
* The SQL engine takes your statements, optimizes them, + compiles them into its processing steps
* *In practice, optimization may happen before or after compilation, or both*
* This is a fair complaint, but a **corollary exists**
* **W/ Spark + other code-heavy processing frameworks, the *code writer* becomes responsible for much of the optimization that is handled automatically in a SQL-based engine**
* The **Spark API is powerful + complex, meaning it is not so easy to identify candidates for reordering, combination, or decomposition**
* When embracing Spark, **DE teams need to actively engage w/ the problems of Spark optimization, especially for expensive, long-running jobs**
* This means building optimization expertise on the team + teaching individual DE’s how to optimize
* A few top-level things to keep in mind when coding in native Spark:
* **1) Filter early and often**
* **Applies to SQL optimization** as well, w/ the difference being that **Spark may not be able to reorder something that SQL would handle for you automatically**
* Spark is a big data processing framework, but the **less data you have to process, the less resource-heavy + more performant your code will be**
* **2) Rely heavily on the core Spark API, + learn to understand the Spark native way of doing things**
* Try to rely on well-maintained public libraries if the native Spark API doesn’t support your use case
* Good Spark code is substantially declarative
* If you find yourself **writing extremely complex custom code**, **pause + determine whether there’s a more native way of doing whatever** you’re trying to accomplish
* Learn to **understand idiomatic Spark** by reading public examples + working through tutorials
* Is there something in Spark’s API that can accomplish what you’re trying to do?
* Is there a well-maintained and optimized public library that can help?
* **3) Be careful with UDFs**
* This is crucial for **PySpark, an API wrapper for Scala Spark**
* Your code pushes work into native Scala code running in the JVM by calling the API
* Running Python UDFs forces data to be passed to **Python, where processing is less efficient**
* **If you find yourself using Python UDFs, look for a more Spark-native way to accomplish what you’re doing** (recommendation 2 above)
* If you *must* use UDFs, consider rewriting them in Scala or Java to improve performance
* **4) Consider intermixing SQL**
* Using SQL allows us to take advantage of the **Spark Catalyst optimizer**, which **may be able to squeeze out more performance than we can w/ native Spark code**
* **SQL is often easier to write and maintain for simple operations**
* **Combining native Spark + SQL lets us get the best of both worlds (powerful, general-purpose functionality combined w/ simplicity where applicable)**
* Much of the optimization advice in this “trade-off’s” section is fairly generic + would apply just as well to Apache Beam, for example
* The **main point is that programmable data processing APIs require a bit more optimization finesse than SQL, which is perhaps less powerful but easier to use**

###### d) Update Patterns

* **Since transformations *persist* data, we will often update persisted data *in place***
* Updating data (DML) is a major pain point for DE teams, especially as they transition between DE technologies
* Remember that the **original data lake concept didn’t really account for updating data**, which now seems **nonsensical** for several reasons
* **Updating data has long been a key part of handling data transformation results, even though the big data community dismissed it**
* It is **silly to rerun significant amounts of work because we have no update capabilities**
* Thus, the **data lakehouse concept now builds in updates**
* Also, GDPR + other data deletion standards now *require* organizations to delete data in a targeted fashion, even in raw datasets
* There are **several basic update patterns**:
* **1) Truncate and reload**
* **Truncate**is an update pattern that **doesn’t update anything + simply wipes the old data**
* In a truncate-and-reload update pattern, a **table is cleared of data, + transformations are rerun + loaded into this table, effectively generating a new table version**
* **2) Insert only**
* **Inserts new records *without* changing or deleting old records**
* Can be used to **maintain a current view of data**
* Ex: If **new versions of records are inserted w/out deleting old records:**
* A query or a view can **present the current data state by finding the newest record by** **PK**
* **NOTE: Columnar databases don’t typically enforce PKs**
* The **PK would be a construct used by DE’s to maintain a notion of the current state of the table**
* The **downside** to the insert-only approach is that **it can be extremely computationally expensive to find the latest record at query time**
* **Alternatively, we can use a materialized view** (covered later), **an insert-only table that maintains all records, + a truncate-and-reload target table that holds the current state for serving data**
* There is an exception to the advice *not* to insert frequently = the enhanced Lambda architecture used by BigQuery and Apache Druid, which hybridizes a streaming buffer w/ columnar storage
* Deletes and in-place updates can still be expensive
* **NOTE: When inserting data into a *column*-oriented OLAP database, the common problem is that DE’s transitioning from row-oriented systems attempt to use single-row inserts**
* This **antipattern puts a massive load on the system**
* It **also causes data to be written in many separate files**, which is **extremely inefficient for subsequent reads, + the data must be re-clustered later**
* **Instead, try loading data in a periodic micro-batch or batch fashion**
* **3) Delete**
* Deletion is critical when a source system deletes data + satisfies recent regulatory changes
* **In columnar systems + data lakes, deletes are more expensive than inserts**
* **When deleting data, consider whether you need to do a hard or soft delete**
* A **hard delete****permanently removes a record from a database**, while a **soft delete****marks the record as “deleted”**
* Hard deletes are useful if there’s a legal or compliance reason to do so or when you *need to remove data for performance reasons* (say, a table is too big)
* *Soft deletes might be used when you don’t want to delete a record permanently but also want to filter it out of query results*
* A 3rd approach to deletes is **closely related to soft deletes: insert deletion****inserts a new record with a *deleted* flag *w/out modifying the previous version of the record***
* This **allows us to follow an insert-only pattern but still account for deletions**
* Just note that our **query to get the latest table state gets a little more complicated**
* We **must now deduplicate, find the latest version of each record by key, and *not* show any record whose latest version shows *deleted***
* **4) Upsert/Merge**
* Of these update patterns, the **upsert + merge patterns are the ones that consistently cause the most trouble for DE teams, especially for those transitioning from row-based DW’s to column-based cloud systems**
* **Upsertingtakes a set of source records + looks for matches against a target table by using a PK or another logical condition**
* *Again, it’s the responsibility of the DE team to manage this PK by running appropriate queries, as* ***most columnar systems will not enforce uniqueness***)
* **When a key match occurs, the target record gets updated (*replaced* by the new record)**
* **When no match exists, the database *inserts* the new record**
* The **merge** **pattern adds to this the *ability to delete records***
* The **problem is that the upsert/merge pattern was originally designed for *row-based* databases**
* In **row-based** databases, **updates are a natural process**, where the **database looks up the record in question and changes it *in place***
* On the other hand, **file-based systems don’t actually support in-place file updates**
* All of these **file-based systems utilize copy on write (COW)**
* **If one record in a file is changed or deleted, the *whole file* must be rewritten with the new changes**
* This is part of the reason that early adopters of big data + data lakes rejected updates, as managing files + updates seemed too complicated
* So, they simply used an insert-only pattern + assumed that data consumers would determine the current state of the data at query time or in downstream transformations
* ***In reality,* columnar databases such as Vertica have long supported in-place updates by hiding the complexity of COW from users**
* They **scan files, change the relevant records, write new files, + change file pointers for the table**
* The **major columnar cloud DW’s support updates *and* merges, although DE’s should investigate update support if they consider adopting an exotic technology**
* There are a **few key things to understand here:**
* **Even though distributed columnar data systems support native update commands, *merges* come at a cost, as the performance impact of updating or deleting a *single* record can be quite high**
* ***On the other hand*, merges can be extremely performant for *large* update sets and may even outperform transactional databases**
* In addition, it is **important to understand that COW seldom entails rewriting the *whole* table**
* **Depending on the database system in question, COW can operate at various resolutions (partition, cluster, block)**
* **To realize performant updates, focus on developing an appropriate partitioning and clustering strategy based on your needs + the innards of the database in question**
* **As w/ inserts, be careful w/ your update or merge *frequency***
* Many DE teams transition between database systems + try to run **near real-time merges from CDC** just as they did on their old system, + it simply doesn’t work
* **No matter how good your CDC system is, this approach will bring most columnar DW’s to their knees**
* Systems can fall weeks behind on updates, where an approach that simply merged every hour would make much more sense
* We **can use various approaches to bring columnar databases closer to real time**
* Ex: BigQuery allows us to stream insert new records into a table, + then supports specialized materialized views that present an efficient, near real-time deduplicated table view
* Druid uses 2-tier storage + SSDs to support ultrafast real-time queries

###### e) Schema Updates

* **Data has entropy and may change without your control or consent**
* External data sources may change their schema, or AppDev teams may add new fields to the schema
* **One advantage of columnar systems over row-based systems is that while updating the data is more difficult, updating the schema is easier**
* **Columns can typically be added, deleted, + renamed**
* **In spite of these technological improvements, practical organizational schema management is more challenging**
* *Will some* ***schema updates be automated?*** (the approach Fivetran uses when replicating from sources)
* As convenient as this sounds, **there’s a risk that downstream transformations will break**
* *Is there a straightforward schema* ***update request process****?*
* Suppose a team wants to add a column from a source that wasn’t previously ingested
* What will the review process look like?
* Will downstream processes break? (Are there queries that run **SELECT \* rather than using explicit column selection, which generally bad practice in columnar databases**?)
* How long will it take to implement the change?
* Is it possible to create a **table fork** (a new table version specific to this project)?
* **A new interesting option has emerged for semi-structured data**
* Borrowing an idea from **document stores, many cloud DW’s now support data types that encode arbitrary JSON data**
* One approach **stores raw JSON in a field while storing frequently accessed data in adjacent flattened fields**
* This **takes up additional storage space but allows for the convenience of flattened data, w/ the flexibility of semi-structured data for advanced users**
* **Frequently accessed data in the JSON field can be added *directly* into the schema over time**
* This approach **works extremely well when DE’s must ingest data from an application document store w/ a frequently changing schema**
* **Semi-structured data available as a 1st-class citizen in DW is extremely flexible + opens new opportunities for data analysts + data scientists since data is no longer constrained to rows + columns**

###### f) Data Wrangling

* **Data wranglingtakes messy, malformed data and turns it into useful, clean data**, + is **generally a batch transformation process**
* Data wrangling **has long been a major source of pain *and* job security for DE’s**
* Ex: Suppose that developers receive electronic data interchange (EDI) data from a partner business regarding transactions + invoices, potentially a **mix of structured data + text**
* The **typical process of wrangling this data involves first trying to *ingest* it**
* **Often**, the **data is so malformed that a good deal of text preprocessing is involved**
* Developers may choose to ingest the data as a single text field table (an entire row ingested as a single field)
* Developers then begin writing queries to parse + break apart the data
* Over time, they discover data anomalies + edge cases
* Eventually, they will get the data into rough shape
* Only *then* can the process of downstream transformation begin
* **Data wrangling tools aim to simplify significant parts of the above process**
* These tools often put off DE’s because they claim to be **no code, which sounds unsophisticated**
* **It’s better to think of data wrangling tools as IDEs for malformed data**
* **In practice, DE’s spend way too much time parsing nasty data, + automation tools allow DE’s to spend time on more interesting tasks**
* Wrangling tools **may also allow DE’s to hand some parsing + ingestion work off to analysts**
* **Graphical data-wrangling tools** typically present a sample of data in a visual interface, w/ inferred types, statistics including distributions, anomalous data, outliers, + nulls
* Users can then add processing steps to fix data issues
* A step might provide instructions for dealing w/ mistyped data, splitting a text field into multiple parts, or joining with a lookup table
* Users can run the steps on a full dataset when the full job is ready
* The job typically gets pushed to a scalable data processing system such as Spark for large datasets
* After the job runs, it will return errors + unhandled exceptions
* The user can further refine the recipe to deal with these outliers
* It’s highly recommended that **both aspiring + seasoned DE’s experiment w/ wrangling tools**
* Major cloud providers sell their version of data-wrangling tools, + many 3rd-party options are available
* **DE’s may find that these tools significantly streamline certain parts of their jobs**
* Organizationally, **DE teams may want to consider training specialists in data wrangling if they frequently ingest from new, messy data sources**

###### g) Example: Data Transformation in Spark

* Let’s look at a practical, concrete example of data transformation
* Suppose we build a pipeline that ingests data from 3 API sources in JSON format
* This initial ingestion step is handled in Airflow, + each data source gets its prefix (filepath) in an S3 bucket
* Airflow then triggers a Spark job by calling an API, + this Spark job ingests each of the 3 sources into a dataframe, converting the data into a relational format, w/ nesting in certain columns
* The Spark job combines the 3 sources into a single table + then filters the results w/ a SQL statement
* The results are finally written out to a Parquet-formatted Delta Lake table stored in S3
* **In practice, Spark creates a DAG of steps based on the code that we write for ingesting, joining, and writing out the data**
* The basic ingestion of data happens in cluster memory, although one of the data sources is large enough that it must spill to disk during the ingestion process
* This data gets written to cluster storage, + it will be reloaded into memory for subsequent processing steps
* The join requires a shuffle operation
* A key is used to redistribute data across the cluster, + once again, a spill to disk occurs as the data is written to each node
* The SQL transformation filters through the rows in memory + discards the unused rows
* Finally, Spark converts the data into Parquet format, compresses it, and writes it back to S3
* Airflow periodically calls back to Spark to see if the job is completed
* Once it confirms that the job has finished, it marks the full Airflow DAG as completed
* *Note that we have 2 DAG constructs here, an Airflow DAG + a DAG specific to the Spark job*

###### h) Business Logic and Derived Data

* **One of the most common use cases for transformation is to render business logic**.
* **This type of transformation happens most frequently in batch transformations**, but note that **it *could* also happen in a streaming pipeline**
* Ex: Suppose that a company uses multiple specialized internal profit calculations
* One version might look at profits *before* marketing costs, + another might look at a profit *after* subtracting marketing costs
* Even though this appears to be a straightforward accounting exercise, *each of these metrics is highly complex to render*
* Profit before marketing costs might need to account for fraudulent orders, + determining a reasonable profit estimate for the previous business day entails estimating what % of revenue and profit will ultimately be lost to orders canceled in the coming days as the fraud team investigates suspicious orders
* Is there a special flag in the database that indicates an order w/ a high probability of fraud, or one that has been automatically canceled?
* Does the business assume that a certain % of orders will be canceled because of fraud even before the fraud-risk evaluation process has been completed for specific orders?
* For profits after marketing costs, we must account for all the complexities of the previous metric, + the marketing costs attributed to the specific order
* Does the company have a naive attribution model—e.g., marketing costs attributed to items weighted by price?
* Marketing costs might also be attributed per department, or item category, or—in the most sophisticated organizations—per individual item based on user ad clicks
* The business logic transformation that generates this nuanced version of profit must integrate all the subtleties of attribution (i.e., a model that links orders to specific ads + advertising costs)
* Is attribution data stored in the guts of ETL scripts, or is it pulled from a table that is automatically generated from ad platform data?
* This above type of reporting data is a quintessential example of **derived data (data computed from other data stored in a data system)**
* Derived data **critics will point out that it is challenging for the ETL to maintain consistency in the derived metrics**
* Ex: If the company updates its attribution model, this change may need to be merged into many ETL scripts for reporting (**ETL scripts are notorious for breaking the DRY principle**)
* Updating these ETL scripts is a manual and labor-intensive process, involving domain expertise in processing logic + previous changes
* Updated scripts must also be validated for consistency and accuracy
* These are legitimate criticisms but not necessarily very constructive b/c the alternative to derived data in this instance is equally distasteful.
* Analysts will need to run their reporting queries if profit data is not stored in the DW, including profit logic
* **Updating complex ETL scripts to represent changes to business logic accurately is an overwhelming, labor-intensive task, but getting analysts to update their reporting queries consistently is well-nigh impossible**
* One interesting **alternative** is to **push business logic into a** **metrics layer, but still leverage the DW or other tool to do the computational heavy lifting**.
* *“Missing Piece of the Modern Data Stack”* (2021): <https://benn.substack.com/p/metrics-layer>
* A **metrics layer encodes business logic and allows analysts + dashboard users to build complex analytics from a *library of defined metrics***
* The metrics layer **generates queries from the metrics + sends these to the database**
* See more about semantic and metrics layers in Chapter 9

###### i) MapReduce

* No discussion of batch transformation can be complete without touching on MapReduce
* Though this isn’t because MapReduce is widely used by DE’s these days.
* **MapReduce** was the defining batch data transformation pattern of the big data era, **still influences many distributed systems DE’s use today, + is still useful for DE’s to understand at a basic level**
* MapReduce was introduced by Google in a follow-up to its paper on GFS + was initially the de facto processing pattern of Hadoop, the OSS analogue technology of GFS (Chapter 6)
* A **simple MapReduce job consists of a collection of map tasks that read individual data blocks scattered across the nodes, followed by a shuffle that redistributes result data across the cluster and a reduce step that aggregates data on *each* node**
* Ex: Suppose that we wanted to run the following SQL query:

**SELECT COUNT**(\*), user\_id **FROM** user\_events **GROUP BY** user\_id;

* The table data is spread across nodes in data blocks, as **the MapReduce job generates one map task per block**
* **Each map task essentially runs the query on a *single* block** (i.e., it generates a count for each user ID that appears in the block)
* While a block might contain hundreds of MB, the full table could be PB in size
* However, **the map portion of the job is a nearly perfect example of embarrassing parallelism**, as **the** **data scan rate across the full cluster essentially scales linearly w/ the number of nodes**
* We then need to **aggregate (reduce) to gather results from the full cluster**
* We’re not gathering results to a single node, but rather we **redistribute results by key so that each key ends up on one and only one node**
* **This is the shuffle step, which is often executed using a hashing algorithm on keys**
* **Once the map results have been shuffled, we sum the results for each key**
* The key/count pairs can be written to the local disk on the node where they are computed
* We collect the results stored across nodes to view the full query results
* Real-world MapReduce jobs can be far more complex than that
* A complex query that filters with a WHERE clause + joins 3 tables + applies a window function would consist of many map and reduce stages

###### j) After MapReduce

* **Google’s original MapReduce model is extremely powerful but is now viewed as excessively rigid**
* It utilizes numerous short-lived ephemeral tasks that read from + write to disk
* In particular, no intermediate state is preserved in memory, as all data is transferred between tasks by storing it to disk or pushing it over the network
* This simplifies state + workflow management and minimizes memory consumption, but can also drive high-disk bandwidth utilization and increase processing time
* The MapReduce paradigm was constructed around the idea that magnetic disk capacity (HDD) and bandwidth were so cheap that it made sense to simply throw a massive amount of disk at data to realize ultra-fast queries
* This worked to an extent, as MapReduce repeatedly set data processing records during the early days of Hadoop
* However, we have lived in a post-MapReduce world for quite some time
* **Post-MapReduce processing does not *truly* discard MapReduce, as it still includes the elements of map, shuffle, and reduce, but it now relaxes the constraints of MapReduce to allow for in-memory caching**
* Recall that **RAM is much faster than SSD and HDDs in transfer speed and seek time**
* **Persisting even a tiny amount of judiciously chosen data in memory can dramatically speed up specific data processing tasks and utterly crush the performance of MapReduce**
* Ex: Spark, BigQuery, + various other data processing frameworks were designed around in-memory processing
* These frameworks treat data as a distributed set that resides in memory
* If data overflows available memory, this causes a **spill to disk**
* The disk is treated as a 2nd-class data-storage layer for processing, though it is still highly valuable
* **The cloud is one of the drivers for the broader adoption of memory caching, as it is much more effective to lease memory during a specific processing job than to own it 24 hours a day**
* **Advancements in leveraging memory for transformations will continue to yield gains for the foreseeable future**

##### Materialized Views, Federation, and Query Optimization

* In this section, we look at several techniques that **virtualize query results by presenting them as table-like objects**
* These techniques **can become part of a transformation pipeline or sit right before end-user data consumption**

###### a) Views

* First, let’s review views to set the stage for materialized views
* **A viewis a database object that we can select from just like any other table, but in practice, is just a query that *references* other tables**
* **When we select from a view, that database creates a *new* query that combines the view subquery w/ our query**
* **The query optimizer then optimizes and runs the full query**
* **Views play a variety of roles in a database**
* 1) Views can serve a **security role**
* Ex: Views can **select only specific columns and filter rows, thus providing restricted data access**
* **Various views can be created for job roles depending on user data access**
* 2) A view might be used to **provide a current deduplicated picture of data**
* If using an insert-only pattern, a view may be used to return a deduplicated version of a table showing only the latest version of each record
* 3) Views can be used to **present common data access patterns**
* Suppose marketing analysts must frequently run a query that joins 5 tables
* We could create a view that joins together these 5 tables into a wide table
* Analysts can then write queries that filter + aggregate on top of this view.

###### b) Materialized Views

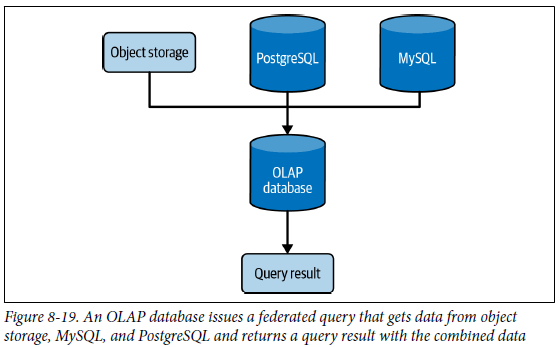
* **A potential disadvantage of (non-materialized) views is that they don’t do any precomputation**
* In the example above of a view that joins 5 tables, this join must run *every time* a marketing analyst runs a query on this view, + the join could be extremely expensive
* A **materialized view does some or all of the view computation in advance**
* In the example, a materialized view might save the 5 table join results every time a change occurs in the source tables
* Then, when a user references the view, they’re querying from the pre-joined data
* **A materialized view is a de facto transformation step, but the database manages execution for convenience**
* **Materialized views may also serve a significant query optimization role depending on the database, even for queries that don’t directly reference them**
* **Many query optimizers can identify queries that “look like” a materialized view**
* An analyst may run a query that uses a filter that appears in a materialized view
* **The optimizer will rewrite the query to select from the precomputed results**

###### c) Composable Materialized Views

* **In general, materialized views do *NOT* allow for composition (they *cannot* select from another materialized view)**
* However, there’s **recently been an emergence of tools that support this capability**
* Ex: Databricks introduced the notion of **live tables** **= each table is updated as data arrives from sources, + data flows down to subsequent tables asynchronously**

###### d) Federated Queries

* **Federated queries**are **a database feature that allows an OLAP database to select from an external data source, such as object storage or RDBMS**
* Ex: You need to combine data across object storage + various tables in MySQL and PostgreSQL databases
* Your DW can issue a federated query to these sources and return the combined results



* Ex: Snowflake supports the notion of external tables defined on S3 buckets (Snowflake **stage**)
* An external data location + a file format are defined when creating the table, but data is not yet ingested into the table
* When the external table is queried, Snowflake reads from S3 + processes the data based on the parameters set at the time of the table’s creation
* We can even join S3 data to internal database tables, which makes Snowflake + similar databases more compatible with a data lake environment
* **Some OLAP systems can convert federated queries into materialized views**
* This **gives us much of the performance of a native table w/out the need to manually ingest data every time the external source changes**
* **The materialized view gets updated whenever the external data changes**

###### e) Data Virtualization

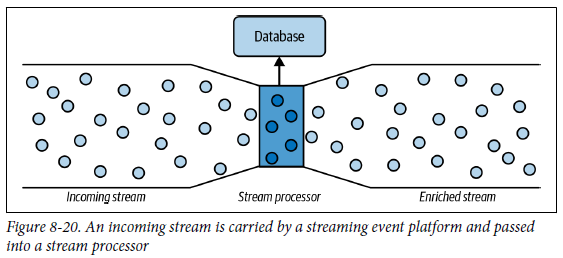
* **Data virtualization**is closely related to federated queries, but this **typically entails a data processing + query system that *doesn’t* store data internally**
* Right now, Trino (e.g., Starburst) and Presto are examples par excellence
* **Any query/processing engine that supports external tables can serve as a data virtualization engine**
* **The most significant considerations w/ data virtualization are supported external sources + performance**
* **A closely related concept is the notion of query pushdown**
* Suppose I wanted to query data from Snowflake, join data from a MySQL database, + filter the results
* **Query pushdown aims to move as much work as possible *to the source databases***
* The engine might look for ways to push filtering predicates into the queries on the source systems
* This serves 2 purposes:
* **1) It offloads computation from the virtualization layer, taking advantage of the query performance of the source**
* **2) It potentially reduces the quantity of data that must push across the network, a critical bottleneck for virtualization performance**
* **Data virtualization is a good solution for organizations w/ data stored across various data sources**
* **HOWEVER, data virtualization should NOT be used haphazardly**
* Ex: Virtualizing a production MySQL database doesn’t solve the core problem of analytics queries adversely impacting the production system
* B/c Trino does *not* store data internally, it will pull from MySQL every time it runs a query
* Alternatively, **data virtualization can be used as a component of data ingestion + processing pipelines**
* Ex: Trino might be used to select from MySQL once a day at midnight when the load on the production system is low
* Results could be saved into S3 for consumption by downstream transformations + daily queries, protecting MySQL from direct analytics queries
* **Data virtualization can be viewed as a tool that expands the data lake to many more sources by abstracting away barriers used to silo data between organizational units**
* An organization can store frequently accessed, transformed data in S3 + virtualize access between various parts of the company
* This fits closely with the notion of a **data mesh**(Chapter 3), wherein **small teams are responsible for preparing their data for analytics + sharing it w/ the rest of the company**
* **Virtualization can serve as a critical access layer for practical sharing**

##### Streaming Transformations and Processes

* We’ve already discussed stream processing in the context of queries, but the difference between streaming transformations and streaming queries is subtle + warrants more explanation

###### a) Basics

* **Streaming queries run dynamically to present a current view of data**, as discussed previously
* **Streaming *transformations* aim to prepare data for downstream consumption**
* Ex: A DE team may have an incoming stream carrying events from an IoT source
* These IoT events carry a device ID and event data
* We wish to dynamically enrich these events w/ other device metadata, which is stored in a separate database
* The stream-processing engine queries a separate database containing this metadata by device ID, generates new events w/ the added data, + passes it on to another stream
* Live queries + triggered metrics run on this enriched stream

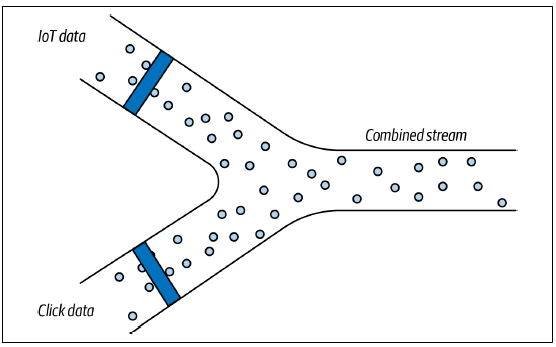


###### b) Transformations and Queries are a Continuum

* **The line between transformations and queries is also blurry in batch processing, but the differences become even more subtle in the domain of streaming**
* *Ex: If we dynamically compute roll-up statistics on windows, and then send the output to a target stream, is this a transformation or a query?*
* Maybe we will eventually adopt new terminology for stream processing that better represents real-world use cases. For now, we will do our best with the terminology we have

###### c) Streaming DAGs

* One interesting notion closely related to stream enrichment + joins is the **streaming DAG** (first talked about in a discussion of orchestration in Chapter 2)
* **Orchestration is inherently a *batch* concept, but *what if we wanted to enrich, merge, + split multiple streams in real time?***
* Let’s take a simple example where streaming DAG would be useful
* Suppose we want to combine website clickstream data w/ IoT data
* This will allow us to get a unified view of user activity by combining IoT events w/ clicks
* Furthermore, each data stream needs to be preprocessed into a standard format



* This has **long been possible by combining a streaming store (e.g., Kafka) with a stream processor (e.g., Flink)**
* Creating the DAG amounted to building a complex Rube Goldberg machine, w/ numerous topics and processing jobs connected
* Pulsar dramatically simplifies this process by treating DAGs as a core streaming abstraction
* Rather than managing flows across several systems, DE’s can define their streaming DAGs as code inside a single system.

###### d) Micro-batch vs. True Streaming

* A long-running battle has been ongoing between micro-batch and true streaming approaches
* **Fundamentally, it’s important to understand your use case, the performance requirements, and the performance capabilities of the framework in question**
* **Micro-batching** is a way to **take a batch-oriented framework and apply it in a streaming situation**
* A micro-batch might run anywhere from every 2 minutes to every second
* Some micro-batch frameworks (e.g., Apache Spark Streaming) are designed for this use case and will perform well w/ appropriately allocated resources at a high batch frequency
* In truth, DBAs and engineers have long used micro-batching w/ more traditional database, but this often led to horrific performance + resource consumption
* ***True* streaming systems** (e.g., Beam and Flink) are **designed to process one event at a time**
* However, this **comes with significant overhead**
* Also, **it’s important to note that even in these true streaming systems, many processes will still occur in batches**
* A basic enrichment process that adds data to individual events can deliver one event at a time w/ low latency
* However, a triggered metric on windows may run every few seconds, minutes, etc.
* When you’re using windows + triggers (hence, batch processing), what’s the window frequency? What’s the acceptable latency?
* If collecting Black Friday sales metrics published every few minutes, micro-batches are probably just fine as long as you set an appropriate micro-batch frequency
* On the other hand, if your ops team is computing metrics every second to detect DDoS attacks, true streaming may be in order
* ***When should you use one over the other?***
* **Frankly, there is no universal answer**
* **The term “micro-batch” has often been used to dismiss competing tech, but it may work just fine for your use case and can be superior in many respects depending on your needs**
* If your team already has expertise in Spark, you will be able to spin up a Spark (micro-batch) streaming solution extremely fast
* **There’s no substitute for domain expertise and real-world testing**
* Talk to experts who can present an even-handed opinion
* You can also easily test the alternatives by spinning up tests on cloud infrastructure
* Also, **watch out for spurious benchmarks provided by vendors**
* Vendors are notorious for cherry-picking benchmarks + setting up artificial examples that don’t match reality (recall “benchmarks” in Chapter 4)
* **Frequently, vendors will show massive advantages in their benchmark results but fail to deliver in the real world for your use case**

#### Whom You’ll Work With

* Queries, transformations, + modeling impact all stakeholders up + down the DE lifecycle
* **The DE is responsible for several things at this stage in the lifecycle**
* From a technical angle, the **DE designs, builds, + maintains the integrity of the systems that query and transform data**
* The DE **also implements data models w/in this system**
* **This is the most “full-contact” stage where your focus is to add as much value as possible, both in terms of functioning systems and reliable + trustworthy data**

##### Upstream Stakeholders

* **When it comes to transformations, upstream stakeholders can be broken into 2 broad categories**:
* 1) Those who **control the business definitions**
* 2) Those who **control the systems generating data**
* When interfacing w/ upstream stakeholders about business definitions + logic, **you’ll need to know the data sources (what they are, how they’re used, and the business logic + definitions involved)**
* You’ll **work w/ the engineers in charge of these source systems + the business stakeholders who oversee the complementary products + apps**
* A **DE might work alongside “the business” + the technical stakeholders on a data model**
* The **DE needs to be involved in designing the data model + later updates b/c of changes in business logic or new processes**
* **Transformations are easy enough to do**; just write a query + plop the results into a table or view
* ***Creating them so they’re both performant + valuable to the business is another matter***
* **Always keep the requirements + expectations of the business top of mind when transforming data**
* The **stakeholders of the upstream systems want to make sure your queries + transformations minimally impact their systems**
* **Ensure bidirectional communication about changes to the data models (column + index changes, for example) in source systems**, as these can directly impact queries, transformations, + analytical data models
* **DE’s should know about schema changes, including the addition or deletion of fields, data type changes, + anything else that might materially impact the ability to query + transform data**

##### Downstream Stakeholders

* **Transformations are where data starts providing utility to downstream stakeholders**
* Your downstream stakeholders include many people, including data analysts, data scientists, MLE’s, and “the business.”
* **Collaborate with them to ensure the data model + transformations you provide are performant and useful**
* **In terms of performance, queries should execute as quickly as possible in the most cost-effective way**
* **What do we mean by useful?**
* **Analysts, data scientists, + MLE’s should be able to query a data source w/ the confidence the data is of the highest quality + completeness and can be integrated into their workflows and data products**
* **The business should be able to trust that transformed data is accurate and actionable**

#### Undercurrents

* The transformation stage is where your data mutates + morphs into something useful for a business
* Because there are many moving parts, the undercurrents are especially critical at this stage

##### Security

* Queries + transformations combine disparate datasets into new datasets
* *Who has access to this new dataset?*
* **If someone *does* have access to a dataset, continue to control who has access to a dataset’s column, row, + cell-level access**
* **Be aware of attack vectors against your database at query time**
* **Read/write privileges to the database must be tightly monitored and controlled**
* **Query access to the database must be controlled in the same way as you normally control access to your organization’s systems and environments**
* **Keep credentials hidden** **(avoid copying + pasting passwords, access tokens, or other credentials into code or unencrypted files)**
* Don’t push database usernames + passwords to GitHub repos nor share w/ other users
* **Finally, never allow unsecured or unencrypted data to traverse the public internet**

##### Data Management

* **Though data management is essential at the *source* system stage (+ every other stage of the DE lifecycle), it’s *especially* critical at the transformation stage**
* **Transformation inherently creates new datasets that need to be managed**
* **As w/ other stages of the DE lifecycle, it’s critical to involve *ALL* stakeholders in data models and transformations + to manage their expectations**
* Also, **make sure everyone agrees on naming conventions that align w/ the respective business definitions of the data**
* Proper naming conventions should be **reflected in easy-to-understand field names**
* Users can also check in a **data catalog for more clarity on what the field means when it was created, who maintains the dataset, + other relevant information**
* **Accounting for definitional accuracy is key at the transformation stage**
* ***Does the transformation adhere to the expected business logic?***
* Increasingly, the notion of a **semantic or metrics layer** that **sits independent of transformations** is becoming popular
* **Instead of enforcing business logic w/in the transformation at runtime, why not keep these definitions as a standalone stage before your transformation layer?**
* While it’s still early days (2022-2023), **expect to see semantic and metrics layers becoming more popular and commonplace in DE and data management**
* Because transformations involve mutating data, it’s **critical to ensure that the data you’re using is free of defects and represents ground truth**
* **If MDM is an option at your company, pursue its implementation**
* **Conformed dimensions + other transformations rely on MDM to preserve data’s original integrity + ground truth**
* **If MDM *isn’t* possible, work w/ upstream stakeholders who control the data to ensure that any data you’re transforming is correct + complies w/ the agreed-upon business logic**
* **Data transformations make it potentially difficult to know how a dataset was derived along the same lines**
* Recall **data catalogs**, **centralized metadata stores for all data across an organization**
* **Often a central place where people can view their data, queries, + data storage**
* **As we transform data, data lineagetools become invaluable**
* **Data lineage tools help both DE’s (who must understand previous transformation steps as they create new transformations) and analysts (who need to understand where data came from as they run queries + build reports)**
* Finally, what impact does regulatory compliance have on your data model + transformations?
* Are sensitive fields data masked or obfuscated if necessary?
* Do you have the ability to delete data in response to deletion requests?
* Does your data lineage tracking allow you to see data derived from deleted data + rerun transformations to remove data downstream of raw sources?

##### DataOps

* **Concerning queries + transformations, DataOps has 2 areas of concern: data and systems (Ops)**
* You ***need* to monitor + be alerted for changes or anomalies in these areas**
* The field of **data observability** is exploding right now, with a **big focus on** **data reliability**
* There’s even a recent job title called “data reliability engineer”
* This section emphasizes data observability + data health, which focuses on the query and transformation stage
* Let’s start with the **data side of DataOps**
* When you query data, are the inputs + outputs correct? How do you know?
* If this query is saved to a table, is the schema correct?
* How about the shape of the data + related statistics such as min/max values, null counts, etc.?
* **Run data-quality tests on the input datasets *and* the transformed dataset, which will ensure that the data meets the expectations of upstream *and* downstream users**
* **If there’s a data-quality issue in the transformation, you should have the ability to flag this issue, roll back the changes, + investigate the root cause**
* Now let’s look at the **Ops (systems) part of DataOps**
* **How are the systems performing?**
* **Monitor metrics such as query queue length, query concurrency, memory usage, storage utilization, network latency, + disk I/O**
* **Use metric data to spot bottlenecks + poor-performing queries that might be candidates for refactoring + tuning**
* If the *query* is perfectly fine, you’ll have a good idea of where to tune the database itself (for instance, by clustering a table for faster lookup performance)
* Or you may need to upgrade the database’s compute resources
* **Today’s cloud and SaaS databases give you a ton of flexibility for quickly upgrading (and downgrading) your system**
* **Take a data-driven approach + use your observability metrics to pinpoint whether you have a query or a systems-related issue**
* The **shift toward SaaS-based analytical databases changes the cost profile of data consumption**
* In the days of on-prem DW’s, the system + licenses were purchased up front, w/ no additional usage cost
* Whereas traditional DE’s would focus on performance optimization to squeeze maximum utility out of expensive purchases, **DE’s now working w/ cloud DW’s that charge on a consumption basis need to focus on cost management + cost optimization (see FinOpsin Chapter 4)**

##### Data Architecture

* **The general rules of good data architecture (Chapter 3) apply to the transformation stage**
* **Build robust systems that can process and transform data without imploding**
* **Choices for ingestion + storage will *directly* impact your general architecture’s ability to perform reliable queries + transformations**
* If the ingestion + storage are appropriate to your query + transformation patterns, you should be in a great place
* On the other hand, if your queries and transformations don’t work well w/ your upstream systems, you’re in for a world of pain
* For example, it’s common to see data teams using the wrong data pipelines + databases for the job
* A data team might connect a real-time data pipeline to an RDBMS or Elasticsearch + use this as their DW
* These systems are *not* optimized for high-volume aggregated OLAP queries + will implode under this workload, so this data team clearly didn’t understand how their architectural choices would impact query performance
* **Take the time to understand the trade-offs inherent in your architecture choices, + be clear about how your data model will work w/ ingestion + storage systems and how queries will perform**

##### Orchestration

* Data teams often manage transformation pipelines using simple time-based schedules (CRON jobs)
* Works reasonably well at first but **turns into a nightmare as workflows grow more complicated**
* **Use orchestration to manage complex pipelines using a *dependency*-based approach**.
* **Orchestration is also the glue that allows us to assemble pipelines that span multiple systems**

##### Software Engineering

* When writing transformation code, you can use many languages (like SQL, Python, + JVM-based languages), platforms ranging from DW’s to distributed computing clusters, + everything in between
* **Each language and platform has its strengths + quirks, so know the best practices of your tools**
* Ex: Might write data transformations in Python, powered by a distributed system (Spark or Dask)
* For a data transformation, are you using a UDF when a native function might work much better?
* Poorly written, sluggish UDFs replaced by a built-in SQL command can sometimes give instant + dramatic improvement in performance.
* **The rise of analytics engineering brings SWE best practices to end users, w/ the notion of “analytics as code”**
* **Analytics engineering** transformation tools like dbt have exploded in popularity, **giving analysts and data scientists the ability to write in-database transformations using SQL, w/out the direct intervention of a DBA or a DE**
* **In this case, the DE is responsible for setting up the code repository and CI/CD pipeline used by the analysts and data scientists**
* This is a **big change in the role of a DE, who would historically build + manage the underlying infrastructure + create the data transformations**
* As data tools lower the barriers to entry + become more democratized across data teams, it will be interesting to see how the workflows of data teams change
* Using a **GUI-based low-code tool**, you’ll get **useful visualizations of the transformation workflow**
* **You still need to *understand* what’s going on under the hood**
* These GUI-based transformation tools will often generate SQL or some other language BTS
* **While the point of a low-code tool is to alleviate the need to be involved in low-level details, understanding the code behind the scenes will help with debugging + performance optimization**
* **Blindly assuming that the tool is generating performant code is a mistake**
* **DE’s should pay particular attention to SWE best practices at the query + transformation stage**
* **While it’s tempting to simply throw more processing resources at a dataset, knowing how to write *clean, performant code* is a much better approach**

#### Conclusion

* **Transformations sit at the heart of data pipelines + it’s critical to keep in mind the *purpose* of transformations**
* Ultimately, DE’s are NOT hired to play w/ the latest technological toys but to **serve their customers**
* **Transformations are where data adds value and ROI to the business**
* It is possible to adopt exciting transformation technologies *and* serve stakeholders
* Chapter 11 talks about the **live data stack**, essentially **reconfiguring the data stack around streaming data ingestion + bringing transformation workflows closer to the source system applications themselves**
* DE teams that think about real-time data as the technology for the sake of technology will repeat the mistakes of the big data era
* But in reality, a lot of organizations have a business use case that would benefit from streaming data
* **Identifying these use cases + focusing on the value *before* choosing technologies + complex systems is key**
* As we head into the serving stage of the DE lifecycle, reflect on **technology as a tool for realizing organizational goals**
* **If you’re a working DE, think about how improvements in transformation systems could help you to serve your end customers better**
* **If just embarking on a path toward DE, think about the kinds of business problems you’re interested in solving with technology**

#### Additional Resources

* *“Data Warehouse: The Choice of Inmon vs. Kimball”* by Ian Abramson
* <https://www.ismll.uni-hildesheim.de/lehre/bi-10s/script/Inmon-vs-Kimball.pdf>
* *“Inmon or Kimball: Which Approach Is Suitable for Your Data Warehouse?”* by Sansu George
* <https://www.computerweekly.com/tip/Inmon-or-Kimball-Which-approach-is-suitable-for-your-data-warehouse>
* “Difference Between Kimball and Inmon” by manmeetjuneja5
* <https://www.geeksforgeeks.org/difference-between-kimball-and-inmon/>
* *“Building a Real-Time Data Vault in Snowflake”* by Dmytro Yaroshenko and Kent Graziano
* *Building a Scalable Data Warehouse with Data Vault 2.0* (Morgan Kaufmann) by Daniel Linstedt and Michael Olschimke
* *“Caching in Snowflake Data Warehouse”* on the Snowflake Community page
* *The Data Warehouse Toolkit* by Ralph Kimball and Margy Ross (Wiley)
* “Data Vault—An Overview” by John Ryan
* “Data Vault 2.0 Modeling Basics” by Kent Graziano
* “A Detailed Guide on SQL Query Optimization” tutorial by Megha
* “Eventual vs. Strong Consistency in Distributed Databases” by Saurabh.v
* “The Evolution of the Corporate Information Factory” by Bill Inmon
* Gavroshe USA’s “DW 2.0” web page
* Google Cloud’s “Using Cached Query Results” documentation
* Holistics’ “Cannot Combine Fields Due to Fan-Out Issues?” FAQ page
* “How a SQL Database Engine Works,” by Dennis Pham
* “How Should Organizations Structure Their Data?” by Michael Berk
* “Introduction to Data Vault Modeling” document, compiled by Kent Graziano and Dan Linstedt
* Simon Kitching
* *“Introduction to Data Warehousing”*
* <https://moi.vonos.net/programming/dwh-intro/>
* *“Introduction to Dimensional Modelling for Data Warehousing”*
* <https://moi.vonos.net/programming/dwh-dimensional/>
* *“Introduction to Data Vault for Data Warehousing”*
* <https://moi.vonos.net/programming/dwh-datavault/>
* Kimball Group
* “Four-Step Dimensional Design Process”
* “Conformed Dimensions”
* “Dimensional Modeling Techniques”
* “Kimball vs. Inmon vs. Vault” Reddit thread
* “Modeling of Real-Time Streaming Data?” Stack Exchange thread
* *“The New ‘Unified Star Schema’ Paradigm in Analytics Data Modeling Review”* by Andriy Zabavskyy
* Oracle’s “Slowly Changing Dimensions” tutorial
* <https://www.oracle.com/webfolder/technetwork/tutorials/obe/db/10g/r2/owb/owb10gr2_gs/owb/lesson3/slowlychangingdimensions.htm>
* ScienceDirect’s “Corporate Information Factory” web page
* *“A Simple Explanation of Symmetric Aggregates or ‘Why on Earth Does My SQL Look Like That?’”* by Lloyd Tabb
* *“Streaming Event Modeling”* by Paul Stanton
* *“Types of Data Warehousing Architecture”* by Amritha Fernando
* US patent for “Method and Apparatus for Functional Integration of Metadata”