Principles of data engineering

s 11C(1)(a)

Contents

Formatted: Tab stops: 15.9 cm, Right,Leader: ...

Introduction

The aim of data engineering is insight. What is insight? Insight is information analysed in the light of interest. Data engineering, therefore, is an activity in which business interest is applied to data to create first information, then insight.

There are other activities besides data engineering that seek insight. These include staff expert elicitation, customer feedback, and market research. In the case of government agencies, these activities include overt, and covert means such as diplomacy and espionage.

Data engineering distinguishes itself from other means of gaining insight through the nature of its methodology. Like other engineering disciplines, it focuses on systematic rigour, efficiency, and reproducibility, and achieve these through the application of scientific and mathematical principles. As with other engineering disciplines, data engineering arose in today's technological society as an effective tool to tackle complexity. The technical nature of data engineering is the subject of this text.

Despite similarities in method, data engineering differs from other engineering disciplines in crucial ways.

First, the data engineer is confronted with the apparent availability of easy substitutes. The business engaging a data engineer is seeking insight, and in many instances, there are seemingly simple alternatives to serious data engineering. For example, business clients may choose to rely on their own judgement or use one of the many available self-service data tools to rapidly create a visually stunning report with little effort. This is not the case in other disciplines. The client who engages a bridge engineer needs a bridge. The client who engages a software engineer needs software. There are no easy alternatives to achieving these aims, and therefore no easy alternatives to the engineering effort.

Second, the search for business insight is open-ended. If a client asks for a bridge, the bridge is the end of the engagement. A bridge is not likely to extend into a longer bridge, nor is the client likely to settle for half a bridge. This is not the case with business insight: a question provokes another question, and a question may be half answered. This interminability of aim means that a data engineering project is likewise open-ended. In no other engineering discipline is the engineer working with such a degree of open-endedness.

Third, the search for business insight is ambiguous. Quite often, when stakeholders begin a data engineering project, they do not know what business insights they are after. Partly this is because business areas first need to discover the information before they know what questions to ask. Partly it is due to the wide-ranging and sometimes conflicting business interests — are we concerned about profit, cost, market share, or negative externalities? And partly it is because of the wide range of stakeholders who

need to be engaged — senior executives and operational staff have valid but different concerns — yet the data engineer is expected to meet all interests. This ambiguity of aim is akin to the challenge of planning a new city suburb to satisfy the needs of all residents. Again, this is a unique challenge for a data engineer.

The availability of easy substitutes, the open-endedness of the search, and the ambiguity of aim are all intrinsic characteristics of the search for business insights. We may loosely summarise these as the **fluid nature** of business insights.

Certainly, the data engineering team can shut its eyes to these problems and insist on hard-and-fast requirements from stakeholders. This approach is unlikely to lead to quality outcomes. Instead, the data engineer's approach must adapt to the fluid nature of business insights.

However, adapting the technical approach alone is insufficient. Instead, the data engineering team has an unparalleled need to combine flexibility, endurance of vision, and the ability to negotiate. Moreover, compared to other engineering firms, the non-technical capabilities sit in a small team – sometimes within a sole data engineer. A data engineering team must confidently wield these non-technical capabilities if they are to achieve excellence in an organisation.

The search for business insights is both demanding in complexity and fluid in nature. This dual aspect makes data engineering one of the most engaging and rewarding disciplines. Success requires both technical and non-technical capabilities. The former is the theme of this text.

Reader's note: This book had been written over a sustained period, in-and-out, and the author had been learning to use LLMs for editing. This makes the prose style uneven and inconsistent. The chapter, Sound judgement, reads especially differently. The content should still be solid, but a second pass for stylistic edit is still to be done. Diagrams also need to be supplied.

Creating information

Expressiveness and fragment modelling

Insight is information analysed in the light of interest. However, the data engineer starts one step further back than information. Data engineering starts with data.

Data comes from Latin for "given." Data is, therefore, what is given with no fixed interpretation, and thus not yet information. Starting with the late 1980's, business discovered that data can be mined as a valuable source of information. Over time, this gave rise to the discipline of data engineering. Data engineering is now one of the most privileged disciplines in the modern era because it is the prerequisite step to convert data to information and become the avenue to business insight.

Data is collected by business processes. This collection can be seen as a projection of the business world into the data world of databases. This projection is often imperfect. First, it can be technically challenging and costly to accurately record business events. Moreover, the business processes making the data record may differ from the business interest of the analyst – for example, when the source of data is not controlled by the business seeking to analyse it. The most important goal of a data engineer is to deal with these imperfections so that the data is ready for business interest.

In this perspective, data engineering is the task of taking data projected by business processes, and re-fashion them into a shape required by business interest.

DIAGRAM: Business world -> Data world -> Business world

The data engineer needs to do this in an efficient, reproducible way for vast amounts of data over a sustained period. Therefore, *the goal of purpose of data engineering is create a cost-effective, robust pipeline of valuable, reusable blocks of information that are conducive to business insight*.

How would the data engineer go about achieving this effectively? By the end of the text, we will introduce a total of five principles. The first two principles are expressiveness and fragment modelling.

Expressiveness

Insight is information analysed in the light of interest. A data engineer who takes this seriously becomes invested in the business world, asks how data relates to it, and is driven by the need to organise the data in a way that makes sense for business decisions. This is the task of applying business interest to reshape data. When done successfully, the output is *expressive* of business interest. The first, and most important, principle of data engineering is therefore the search for expressiveness.

Expressive entities refer to the idea that the tables and relationships we create in the data warehouse should not merely reflect the data as we found it but corresponds strongly to real business processes. It is expressive because the consumer or reviewer of the model can clearly recognise the world the model is attempting to approximate. If

a reasonably competent layperson cannot easily relate the model to the real world, then the model is not expressive. Correspondence to the world by the agent trying to influence it is therefore the final arbitrator of whether a data model is successful.

Expressiveness encompasses all aspects of a good data engineer's work. Expressiveness lies in everything – from the way the engineer design tables, choose to name tables and columns, to the way they are connected by primary and foreign keys, right through to the way the engineer describes the intent of code to others.

Fragment modelling

The fluid nature of business insights means that the data engineer cannot know in advance what needs to be built. At a micro level, this applies to a single data engineering project. At a macro level, discovering business insights can be a journey of years over multiple teams. Moreover, in a large organisation different teams will want to see the same information in different ways to reflect specific business interests.

These explain why the goal of a data engineer is to produce reusable blocks of information rather than solely on building complete data products. When the data engineer shifts focus from building complete products to creating blocks that can be reused for multiple products, this will lead to a shift of seeing the warehouse as a repository of data models to seeing it as a fragment store of valuable information. A data engineering approach focused on a creating a store of valuable fragments is fragment modelling.

Modularity of code is a cornerstone in software engineering. Fragment modelling takes this concept and applies it to the modularity of information creation in a data pipeline. The result, as in software engineering, is a benefit of clarity, flexibility and reusability in information. The data engineer, therefore, sees the pipeline primarily as software code and tables as functions. In fact, much of the wisdom in *the Zen of Python* applies carries to fragment modelling.

Fragment modelling is concerned not only with the informational value of a fragment, but also on its intended use. The data engineer does not rest until the fragment becomes plug-and-play for the end-user who can make use of the information with minimal additional engineering. In most cases, this means designing fragments that integrate cleanly into traditional dimensional models for self-service business intelligence. However, it also means ensuring that fragments can be adapted for other analytical scenarios such as data science feature engineering. This combined focus on both information and usability makes fragment modelling especially suited to the demands of the data age where data analytics is employed in a growing range of contexts.

Fragment modelling is also an important step towards expressiveness. A wide table with large number miscellaneous columns buries multiple business concepts into one

undifferentiated “thing” that does not recognisably map to any real-world entity. On the other hand, tightly defined fragments bring clarity to business attributes by giving them sharp relief. It is impossible to achieve a truly expressive data model without deconstructing source data into meaningful fragments and re-organising them to better reflect real-world entities.

Fragment modelling can be disorienting for an engineer used to traditional warehousing approaches. Once familiar, it becomes a powerful approach to creating sustainable and flexible pipelines. The section “Creating information” will explain how this is done.

s 22(1)(a)(ii)

Mapping the data world

Business is not interested in data for data's sake. It is interested in the business reality behind the data. From this perspective, data is not an end in itself. Its function is to bridge the business processes that collect the data and the business interest that seeks insight.

Consequently, the data engineer sees each data record as an intermediary of the business entity that generated it. A business entity can be a concrete object, such as a customer or a product. It can also be an abstract event, such as a business transaction.

For a data engineer, the shift from focusing on data itself to focusing on the business entity behind the data is a prerequisite mental leap. Today, there are various pathways into the data engineering discipline. Some come from business areas and pick up data engineering to answer questions they have about the business. As data grows in social importance, more individuals enter through academia. The danger for the latter is that they are more prone to treating data and data techniques as ends in themselves and may lose valuable years before making the necessary shift in thinking—if they ever do so at all. Academic training can be an excellent starting point as it builds foundational technical expertise. All that is needed is to make the mental leap early in the journey as a data engineer.

For a data engineer who starts with the data and is reaching back to the business world, the tool of choice is the primary key.

Primary key

Primary keys play a special role in establishing the correspondence between the business entities in the real world and data records in the data world. A primary key is what business processes use to accurately retrieve a business entity's data record from a data store.

For example, an order number identifies a sales transaction, and a customer number identifies a client record. These keys are visible to end users who use by them to relate back to the business processes, such as calling up the company to follow up on an order. On the other hand, systems often employ internal keys, invisible to users, to support subsidiary information retrieval. In every case, the function remains the same: reliable lookup of records.

The quality of a primary key is underpinned by its business process, which may be:

- **Rigorous**: A strict identity-proofing protocol (e.g., issuing social security numbers) ensures unique assignment of key to the personal identity. Mistakes can occur. However, the overall system remains robust across the large population.

- **Fuzzy**: Smaller organisations may register customers by phone number or email. When the database is small, this would return accurate results. However, increased number of customers may lead to duplicate records or accidental access if contact details overlap.
- **Log-like**: Modern online shops often implement a guest checkout to cater to buyers' preference. Guest-checkout creates a new record for every transaction. In this case, we designate these as a *log-like* table because every event becomes logged.

The final example is a case of having no primary keys. This lack of primary key could be by design as in the example of a guest checkout. The data engineer can treat each row as its own entity and assign a random, unique number for primary key. If business needs to know the underlying business entity driving each record, such as the identity of the guest shopper, then the data engineer or scientist may apply entity resolution techniques to deduplicate the data.

The lack of primary keys can also be by accident. For example, it could be a business process where staff records information per day on a spreadsheet, without a clear key for each row on the spreadsheet to identify how they relate to business entity.

Ultimately, the ability of a primary key to serve as a reliable link to the real world is directly proportional to the quality of its lookup process. Implementing and maintaining a strong primary-key process carries cost. Quite often the cost becomes a barrier to maintaining an accurate correspondence between the data records and the business entity, resulting in duplication or loss of information:

- When record creation is easier than updates, duplicate entries proliferate.
- When creation or updates become burdensome, entities may go unrecorded or existing records may be overwritten or repurposed, resulting in information loss.

Any primary-key issue inevitably traces back to flaws in the underlying business process. If there are issues with the primary key, the data engineer must do more than accepting the issue at face value but work to find the root cause in the business process.

Given its foundational role, the primary key is the natural starting point for a data engineer examining a new system. Often the database does not come with keys implemented as constraints, for example, if uniqueness of data is implemented in the application layer. In these cases, a data engineer investigating a new system would do well with inferring the primary keys for each table, and consequentially, asking how the primary keys are created, retrieved, and deleted in business processes.

A direct corollary of the primary key's significance is that relational algebra becomes indispensable: experienced engineers think in keys and rely on algebraic operations to track and communicate data rather than memorise every detail of source processes. Discussion of relational algebra lies beyond this text's scope - literature abounds.

Immutable and mutable entities

It is helpful to divide business entities into two broad types — immutable and mutable.

Immutable entities are those that do not change. If they do change, the change is treated as the creation of a different entity, and the latest version is considered the "correct" one. In other words, neither the entity nor its attributes evolve over time. Examples include a completed bank transfer or an importer lodging an import declaration into a country. If the import declaration is altered, it is regarded as a new entity. Products can also be treated as immutable. A new model of phone is considered a different product from the previous version.

s 22(1)(a)(ii)

Mutable entities are those that may change over time while still being considered the same entity. A customer may change purchasing habits, or an employee in an HR system may update personal details or gain new qualifications. These changes are important but, according to business interest, still part of the same entity.

This distinction is helpful because most business processes are built with one of these perspectives in mind. A digital system may have sub-systems that manage both. For example, a sales system may record purchases as immutable entities, treat products themselves as immutable, but also register information about customers whose attributes and behaviour are mutable.

One way to distinguish mutable from immutable entities is the presence of a datetime component in the primary key to indicate the validity period of the record. In warehousing terminology, such a record is a Type II record. The datetime signals that it is important to track changes to the entity's attributes over time. This tracking may be implemented in the source system and maintained by business processes, or it may be absent and later added by a data engineer to meet business interest.

The two major types of business entity — immutable and mutable — correspond to two major types of data engineering methods. These are entity processing and entity tracking.

Entity processing and entity tracking

Zoomed out overview of both

Entity processing **Entity tracking**

Formatted: Font: Bold

	<Description>	<Description>
Most suitable to use when modelling..	Immutable entities.	Mutable entities.
Common characteristics of these types of problems include...		
The usual goals of the data model are...		

Entity processing

The most common pattern in data engineering work is that of entity processing. This applies to immutable business entities — those that do not change once created. Examples of immutable entities include a sales order, a bank transfer, or an import declaration. If they do change, it would be considered sufficiently different that it is a different entity. An example would be the lodgement of an import declaration. If the content of the declaration changes, it would be justifiable to consider it to be different declaration where the earlier version is invalidated. For immutable entities, the business objective is to process them according to defined business logic, hence the term entity processing.

Immutable entities often share certain characteristics:

- **Versions**. Each version can be considered as a new entity and the latest one replaces all others as the "correct" version.
- **Header and details structure**. The header represents the main entity and the details represents its components. Example would be a sales order and line items within the order.
- **Subprocesses**. Multiple business processes are often needed to fully complete the business activities on the entity. Example include sales, which operates on the entire basket of goods, and sales item refunds, which refunds items from a previous sales.
- **Good or bad**. Immutable entities often come with a concept of "good" or "bad" entities, as in the case of customer sentiment, fraudulent transactions, or non-compliant import of goods. Quite often this property is not within the application system itself, but is a layer of interpretation added after the fact from the perspective of business interest.
- **Actors.** There may be actors who persist over time and over multiple immutable entities. Examples include a customer making multiple purchases over time or an importer making multiple import declarations over time.

The pattern of entity processing is suited to working with immutable entities. The usual goals of the entity processing pattern are to:

- Obtain insights about the entities from underlying processes.
- Identify macro-level trends on the business entities such as the level of sales per region. If there is a concept of good or bad entities, then there will be particular interest in their changing trends.

- Measuring the time to reach process milestones defined by the business. In particular, the time from the entity's entrance into and exit from the business process is the total time to process.
- Provide drill through to information about specific events that took place, usually to troubleshoot problems or to identify the root cause of bad entities.
- Provide a foundation for predictive analytics by identifying attributes that may be indicative of bad entities. This often done as a part of feature engineering for machine learning.
- Provide a basis for understanding the actors and how their behaviour affects business outcomes.

The first task is to identify the entity of interest. Most often this entity is clear, for example, the entity of interest in a business process to screen international travellers for communicable disease will be the traveller. Sometimes the entity of interest can be hard to identify. For example, a sale transaction may have multiple line items. Depending on the nature of the business, the entire transaction may be the entity of interest, or the line item themselves would be the preferred entity of interest. The lens to adopt is determined by what makes sense for business action – are we targeting more sales events through sale campaign, e.g. Amazon sales, or focused on creating the next phone version to drive sales, e.g. the next iPhone? Focusing on sales campaign may drive towards treating the entire sales order as one entity without putting weight into the check out basket, which in the case of Amazon, are likely to be basket of heterogeneous basket of goods. Focusing on the product itself, such as the iPhone version, may drive towards treating each line item as the entity of interest. Ultimately, the data engineer may need to adopt multiple perspectives for different scenarios.

Once the entity of interest is identified, the data engineer's aim is to tell the high-level story about the entity of interest while maintaining all the necessary detail. Fragment modelling is especially suited to this task because the information about these different aspects can be maintained in different fragments.

Building the pipeline

A data engineer building a pipeline for entity processing can follow three broad steps:

1. **Filter** — remove irrelevant data and focus on the minimum blocks of information
2. **Map** — compute reusable blocks of information
3. **Reduce** — aggregate to the level required for analysis

First pass — Filter

In this first pass, the data engineer works predominantly with incoming raw data. The aim is to establish structure and meaning, starting with the identification of keys. The

first pass itself has three stages – identifying the key, defining reference tables, and extracting transactions.

Identifying keys

The first task is to identify the business keys that serve as primary keys to the raw data. These may be defined as database constraints in the source data, or they may need to be inferred through exploratory analysis. Defining primary and foreign keys is a simple but powerful way to bridge the gap between the data world and the business world. Without keys, the data appears as a miscellaneous collection of rows. With keys, the data begins to reflect real-world entities and relationships.

In some cases, key columns may need to be created. For immutable entities, there are two common scenarios — versioning and sequence numbers.

In versioning, a typical application might use a column like [Sales ID] and another like [Previous sales ID] to indicate that one record replaces another. A clearer approach is to introduce [Original sales ID] and [Sales version number], using partitioning and ordering to establish lineage. The original [Sales ID] can still be retained for joins but should be renamed to [Sales version ID]. Together, these columns communicate that a sales entity can have multiple versions, and that the latest version may be the one of interest. This is far more expressive than relying on [Sales ID] and [Previous sales ID] alone. The column [Previous sales ID] can be dropped for clarity.

In header–detail structures, it's common for detail rows to be stored as simple lists. For example, a [Sales ID] may be associated with multiple [Sales item ID] entries. It may be appropriate to introduce a [Sales item sequence number] to clarify the relationship between the order and its line items. The full primary key would then be [Sales ID], [Sales item sequence number], and the original [Sales item ID] can be retained for joins.

Keys do not need to be enforced as physical constraints. It suffices to store them in separate metadata tables. What matters is that they communicate meaning and can be looked up by consumers of the data.

Define reference tables

Reference tables are small, slow-moving, descriptive tables that describe the content of larger, fast-moving transaction tables. Examples include *RefProduct*, which describes the products being sold, and *RefLocation*, which describes the store where the sale occurred.

Reference tables are important because they are:

1. Expressive — they provide a central place to describe the attributes of a business entity, rather than embedding all information in the transaction table
2. Efficient — updates can be made to a small reference table rather than to large volumes of transaction data

The data engineer can use reference tables from source systems for inspiration, but should not be limited by them. Application reference tables are often designed to support system functionality, not business insight. There are several ways to add value and make reference data more expressive.

First, rename incoming tables and columns to reflect business interest. Application tables often have poor naming because the UI handles presentation. The underlying data should be renamed with care and clarity.

Second, application reference tables are often normalised for retrieval efficiency, which can lead to fragmentation. It may be useful to combine several reference tables into one. For example, *RefProductSource*, *RefProductDescription*, and *RefProductPriceRange* may be merged into a single *RefProduct* table to clarify the concept of product.

Third, the data engineer can create new reference tables from scratch. Applications rarely include analytical reference tables, but these can be valuable for business insight. For example, while the source system may describe sales using *RefProduct* and *RefLocation*, it may lack a *RefSales* table. A custom *RefSales* table could include attributes such as sales season (inferred from the calendar), a binary flag for whether a discount code was applied, or any other attributes related to the sales order. These additions promote expressiveness and help the business understand drivers of performance.

Another difference between reference tables for analytics and those for applications is the use of unknown or default values. Applications often do not need a row for missing values because the UI handles presentation, and transactional consistency prevents gaps. This is not the case with analytical reference tables. For these reasons, reference tables may need to be supplemented with a default such as “Unknown product.”

Reference data is also the right place to identify conformance across the warehouse. Is there already a *RefLocation* used in another domain? If so, aligning with it can support integration. This applies to more abstract concepts as well, such as *RefProcessingStatus*. If multiple business processes share similar stages, they can be reshaped to use common statuses for an enterprise-wide view. Doing this consistently maximises the power of the warehouse.

Finally, because reference table describe attributes of business entities, they are well-suited to include metadata such as business definitions, descriptions, and other annotations that express meaning. If there are scripts for each table, then the metadata can be written in the scripts of the reference table. If the organisation uses a third party tool for cataloguing, then the reference table catalogue entry is the place to document the metadata.

Extract transaction tables

Transaction tables are fast-moving tables that record business events. They are the primary source of information value for the business.

When extracting from transaction tables, use a name that pairs with a reference table that describes the transaction. For example, *Sales* should be matched with *RefSales*, *Inspection* with *RefInspection*, *AuditResult* with *RefAuditResult*, and so on. A transaction table will often have multiple reference tables to describe it, such as *RefProduct*, *RefLocation*, etc. Pairing the transaction table with a reference table of the same name signals to the user where to focus for context and understanding. In some cases, the application may already have a table like *RefInspectionType*, which can be renamed and used as *RefInspection*. In other cases, a main reference table may need to be created from scratch.

When extracting, filter out noisy data such as unnecessary columns, and where justifiable and computationally simple, unnecessary rows. The purpose of filtering is not just efficiency—it is also to remove noise and improve clarity for both the data engineer and the business. This filtering needs to be balanced with retaining important information. It may be acceptable to filter out fake testing data that has made it to production, but it is usually best to keep all versions of an entity rather than just the latest valid version.

An important case for filtering is to deal with transactional inconsistency. During batch extraction, it is common for inconsistency to arise within the batch. For example, a detail row may be loaded without its header, such as a sales item without a sales order. Depending on the business use, this inconsistency can be significant, and early records should be filtered out until the next batch. If this is done, it is important to ensure the next round of extraction picks up the skipped records.

The transaction table should be as narrow as possible—that is, with a minimum number of columns. This can be achieved by passing as much information as possible to reference tables. For example, rather than extracting location name and region into the transaction, keep only the [Location ID] and pass the attribute information to *RefLocation*. This may require creating new reference tables. It is common for applications to store multiple “Yes/No” columns in the transaction table. Suppose there are four such columns—this generates 16 combinations. These can be expressed in a single expressive reference table, and the four columns in the transaction table can be replaced with a single foreign key. This keeps the transaction table narrow while promoting clarity of meaning.

Extracting the transaction is also the right place to backfill missing values with defaults where applicable. If the reference table includes a default value, then the backfill should use the primary key of that default row.

When these steps are applied, the data engineer will have:

- Extracted the minimum amount of information—not just for efficiency, but to remove noise that impedes clarity
- Built a basic map of information by implementing primary and foreign key relationships, and added quality metadata
- Created expressive meaning through clear naming, especially by relying on reference tables
- Applied change detection to incoming records to support downstream change tracking (see the chapter *Load Mechanics*)
- Set up a foundation for future optimisation work, including incremental extract

Second pass — Map

The first pass focuses on extracting raw data, applying basic transformation to clarify meaning, and setting up the foundations for deeper analytical work. The second pass builds on these foundations. It focuses on creating useful information—derived, calculated, and shaped for business insight. This pass avoids working directly off the raw data and instead uses the outputs from the first level of processing.

The mindset here is that of storytelling. There is a later chapter dedicated to this topic. The chief consideration is not to be limited by the source system concepts, or even the business terms provided by stakeholders. The data engineer must be appropriately creative in reshaping the language in a way that makes sense for business decision-making.

In creating information, the guiding question is: “What are the valuable pieces of information that would be useful for business insight?” Often, this information is latent in the data and needs to be explicitly calculated. For example, the concept of a non-compliant transaction may not be stated in the source data and must be inferred. This applies generally to the idea of good or bad entities. Application systems are rarely designed with these definitions in mind—they are often applied afterwards by analysts as a matter of business interpretation.

The sky is the limit for what sort of information can be usefully created. A fuller sample of possibilities is covered in the chapters *Storytelling* and *Meaningful Fragments*.

When creating information, the result should be stored in standalone tables rather than as additional columns in the extracted tables. This is in line with the concept of fragment modelling and provides a clean container for complex logic. This modularity is why the creation of information is done as a second pass, rather than mixing it with the first pass of extraction.

As in the first pass, information should be offloaded to reference tables, and the transactional tables kept narrow. This may require the creation of new reference tables from scratch as it is a concept that does not exist in the source data.

As an example of the first pass and second pass, if the first pass created *Sales*, *SalesItem*, *ItemSupplierCost*, *SalesItemRefund* from the raw data, the second pass may use these to create a table called *SalesItemMargin* to compute the profitability on each sales item.

The full set of tables in this example would look like:

1. First pass from the source data
   - *Sales* – header for the sales event
   - *SalesItem* – a list of items sold in the sales and for how much
   - *ItemSupplierCost* – how much does the item cost to supply, a type II table since the supply cost can change
   - *SalesItemRefund* – a separate table that record any refunds
2. Second pass of derived computation
   - *SalesItemMargin* — computes the profitability on each sales item after calculating the sale item value, refunds, and costs.

Third pass — Reduce

The third pass focuses on aggregation. At this stage, the data engineer takes the detailed fragments from the first and second passes and rolls them up to the grain that matches the entity of interest. This is important because source systems often contain large volumes of fine-grained detail, yet the business focus is centred on the entity itself. Decision makers want to understand the final sale and not every sub-event in isolation. The role of the data engineer is to distil those details into a clear, entity-level view.

This stage is also a major opportunity for a data engineer to add value. Aggregations can be technically complex, especially when they involve multiple sources or complex business rules. Many business teams find it difficult to perform these calculations consistently and accurately. By building them into the pipeline, the data engineer delivers ready-made, trusted information blocks that can be used without the need for repeated manual work.

As in the two previous passes, reference tables should be the focus of expressiveness and documentation. They will need to be created, and information should be passed to reference tables whenever possible. More details are covered in the chapter *Storytelling*.

Continuing the earlier example, suppose the first pass produced *Sales*, *SalesItem*, *ItemSupplierCost* and *SalesItemRefund*, and the second pass created *SalesItemMargin*

to calculate the profitability of each sales item. In the third pass, this information can be aggregated to the sales level to produce *SalesProfit*. This table would combine the total margin from all items in the sale, apply any additional business rules to arrive at a final profit figure for the sale. The profit at the sales level may be positive or negative. A main reference table, *RefSalesProfit*, could include a binary flag indicating whether the sale was profitable. When coupled with other attributes from *RefSales, RefLocation, etc* they begin to tell a story about the company's sales, such as identifying patterns in unprofitable transactions or highlighting the proportion of sales that meet profitability targets.

This pass is also the place to compute the time where an entity reaches specific milestones. For example, a sales order may have a milestone for placement of the sale, confirmation, shipment, and completion. These dates can be recorded through columns of *SalesProcessMilestone*. This computation is in the reduce stage because a business process often performs a specific checkpoint multiple times, and it is often the earliest or the latest of such instance that matters, requiring aggregation. For example, a sales order may split into multiple shipment.

The outputs of this pass are valuable building blocks. In business intelligence scenarios, they provide ready-to-use measures for dashboards and reports, removing the need for analysts to rebuild complex aggregations. In data science scenarios, they are equally important. Much of feature engineering in machine learning involves aggregating detailed attributes back to the correct grain. A well-designed *SalesProfit* table can feed directly into models that predict customer lifetime value, sales team performance or the likelihood of repeat purchases. By completing this third pass, the data engineer ensures that the pipeline delivers insight-ready, entity-level information that can drive decisions across the organisation.

Summary of the three passes

When complete, the pipeline should create a reusable block of valuable information, expressed in a way that reflects business interest, supported by good metadata, and delivered through an efficient and robust pipeline. These blocks of information are immediately for different purposes. They are ready for presenting in a self-service dimensional model.

In addition, much of data science work for feature engineering is to roll information back up to the entity level for machine learning. Since the data engineer starts with defining the entity of interest and completes the pipeline with correct aggregation, the outputs are also ready for feature engineering.

Common problems

New data engineers benefit from being aware of common mistakes in entity processing.

s 22(1)(a)(ii)

Starting with the details. When developing an entity processing pipeline, the order of work should follow the order of the business process itself. The starting point should be the main entity — for example, *Sales* — before moving on to its associated detail tables such as *SalesItem* or *SalesItemRefund*. The temptation to begin with the details should be resisted. In some business areas, such as those focused on customer satisfaction, there may be pressure to start with a specific detail like refunds. However, without first establishing the context of the header entity, the detail cannot be properly understood or related back to the business process.

Neglecting reference tables. A common mistake, especially for those new to the discipline, is to store too much information in the transaction table rather than keeping it narrow and letting reference tables carry the descriptive load. cokmtables are designed to hold the attributes that matter to business interest. Their consistent use promotes expressiveness and efficiency, and helps maintain a clear separation between event data and descriptive context.

"Garbage in, garbage out". Many application systems have poor or cryptic names for tables and columns because presentation is handled in the user interface and naming is not a primary concern. New engineers who are not confident in creating or naming new concepts may fall back on these application names. This is a missed opportunity to apply the principle of expressiveness. Renaming and reshaping to reflect business meaning is one of the most direct ways to make the model recognisable and useful.

Skipping aggregation. The third pass exists to aggregate information to the entity level. New engineers can often stop at the first or second pass without completing this step. This is partly because many business areas will see the information from the earlier passes and be satisfied with the data they recognise, without realising the value of aggregating it to a level that supports decision making. Stopping at the detail level without rolling information back up to the entity of interest is a missed opportunity if the business never realises the value of the aggregation. On the other, the business may be left to perform complex aggregations themselves, forfeiting one of the most valuable contributions a data engineer can make.

Compounded logic. Placing too much computation, sometimes unrelated, into a single table makes it computationally unwieldy and difficult to maintain. The principle of fragment modelling suggests breaking these computations into standalone tables. This keeps logic modular, makes testing easier, and allows fragments to be reused in different contexts without carrying unnecessary baggage.

More specifically, new engineers often mix the first and second passes into one. The first pass focuses on the raw tables and filters out unnecessary data. The second pass builds on top of these cleaned and structured outputs to create new information. Combining the two means extracting data and performing complex transformation in

the same computation. This compounds computational complexity and undermines future possibilities for incremental extraction due to complex query logic.

In general, there is a temptation to do all three passes at once in the beginning because of convenience. In practice, this creates a host of problems that fragment modelling avoids.

Avoiding these mistakes keeps the focus on the business outcome and ensures that the pipeline remains clear, adaptable, and valuable over time. By grounding the work in the main entity, keeping reference tables central, and maintaining a disciplined modularity in computation, the resulting pipeline will stand the test of time and serve as a reliable foundation for both business intelligence and analytical use.

Entity tracking

The second most common pattern in data engineering work is that of *entity tracking*. This pattern applies to mutable business entities — those that can change over time while still being considered the same entity. Examples include a customer whose contact details, preferences, or purchasing behaviour evolve; an employee whose qualifications, role, or performance record changes; or a supplier whose compliance status shifts over time.

Mutable entities often share certain characteristics:

- **Status**. They have no clear "versions" in the sense used for immutable entities, but they do have *statuses* that change over time.
- **Changing attributes**. They possess multiple detailed attributes, and changes to these attributes define the entity's mutation. For example, in an HR system, an important attribute in an employee is a qualification record, where the acquisition, expiry, or renewal of a qualification is a key business event.
- **Actions**. They are often actors in business processes, and their performance or behaviour is of ongoing concern. Their behaviour is often what defines a "good" versus a "bad" entity.
- **Audit**. They may be the subject of additional business processes that *audit* them periodically or in response to events. The result of the audit may also define whether they are good or bad.
- **Registration**. They are typically underpinned by an *entity registration* process, which establishes their initial presence in the system, and serve as entry points for ongoing mutation of attributes. For example, a client sign-up form will register a client in an online shop.

The usual goals of entity tracking are to:

- Monitor macro-level trends in the number of entities over time, including trends of the composition of the population, or trend in number of good or bad entities.
- Track the management activities associated with these entities, such as registration and audit.
- Monitoring when entities reach performance milestones defined by the business thresholds. For example, when an employee has met enough sales target.
- Detect anomalous entities by some measure — often a good or bad classification as inferred from their behaviour or audit result — for targeted action and early intervention.

- Provide drill-down information about the entity and the changes it has undergone, usually for troubleshooting.
- Understand the effect of entities, which are often actors, on other business outcomes, such as the qualification of employees on the ability to meet the organisation's delivery targets.

Two ways to track

There are two common approaches to tracking an entity. The first is by its attributes; the second is by its behaviour.

Tracking by attributes means that an entity is registered into a database, and there are tables which record its attributes such as name, address, status, and so on. These tables are usually type II tables—that is, each row is labelled with its validity period using columns like [Start date] and [End date]. A common scenario is tracking an employee's history in an organisation: their position, pay, and other slowly changing properties. Systems with a strong presence of attribute tables, especially type II, are clearly entity tracking problems. Here, the challenge for the data engineer is to present the evolving attributes to the user coherently. This becomes more complex when multiple attributes are tracked across different tables.

Behaviour informs performance. The other approach is to track an entity is by its behaviour. In this case, the business is interested in insights about the entity through observing the way it behaves over time. There are two ways for a business to observe an entity's behaviour: through audits of its state or through records of its actions. For example, in the case of an employee, an audit might be an independent assessment of their output, while actions might include records of the staff daily work. Both provide vital information about the business entity that needs to be tracked.

Systems that track behaviour are less obviously entity tracking problems because they may lack a type II primary key on attributes to indicate mutability. In these cases, the tracking requirement is known through business intent. The data engineer's role is to build an accurate representation of the entity's behaviour and work with business users to apply rules that infer performance from that behaviour.

Many systems have elements worth tracking. For example, a bank may want to track its account holders. This can be done in two ways:

- **Attribute**. Bank accounts have evolving attributes such as account holder, account type, and account level (e.g. silver, gold, platinum). These attributes change infrequently but are useful for assessing the account's current and potential business value.

- **Behaviour**. Deposits, withdrawals, transfers, and interest accruals are recorded as actions taken by the account holder. These behaviours also help reveal the account's current and potential business value.

Building the pipeline to track attributes

Much of the approach from *Entity Processing* applies. The same practices—identifying the entity of interest, crafting expressive reference tables, maintaining detail, and telling a coherent story—remain central. The difference is that for mutable entities, the pipeline has additional task of tracking attribute changes, which introduces new techniques.

Consider the example of a bank tracking its account holders. The core table is *Bank.Account*, which contains evolving attributes such as account type, account level (e.g. silver, gold, platinum), and account status (e.g. active, dormant, closed). These attributes may change over time, and the business is interested in understanding how accounts evolve.

First pass — Filter

The first step is to remove noise, build a semantic map of the basic information, and create reference tables to support meaningful context. Compared to entity processing, entity tracking introduces an additional step for tracking attributes: **temporal compression**.

Temporal compression collapses consecutive rows that carry identical information across time periods. It is akin to run-length encoding. For example, consider the *Bank.Account* table with primary key [Account ID], [Start date], and attributes like [Account type], [Account level], and [Account status]. If these attributes remain unchanged across multiple rows, they can be compressed into a single row representing the full validity period.

Example before compression:

Account ID	Start date	End date	Account type	Account level	Account status
12345	2022-01-01	2022-06-30	Savings	Silver	Active
12345	2022-07-01	2023-06-30	Savings	Silver	Active

After compression:

Account ID	Start date	End date	Account type	Account level	Account status
12345	2022-01-01	2023-06-30	Savings	Silver	Active

Implementation varies by technology. It becomes more complex when supporting incremental extraction, especially if changes are not at the end of the batch or if deletions are possible.

In addition to compression, the first pass should assign a ***surrogate key*** *to each validity period. For example, [Account SK] would represent the combination of [Account ID] and [Start date]. This surrogate key becomes the anchor for downstream joins and timeline construction. Ideally, this is handled automatically by warehouse automation.*

Second pass — Compute the timeline

An entity usually has attributes stored in multiple tables. For immutable entities, retrieving attributes from multiple tables into one result set is a simple matter of joining on the appropriate keys. This is not the case for mutable entities. In a mutable entity, each of the attribute tables are type II, that is, each row is defined by a validity period. Joining attributes in this scenario means taking these into account. It is a matter of solving what is known as the *overlapping window* problem. This can be challenging for business users, and is easy to get wrong when two, three or even more attributes are involved.

The second pass is to simplify this complexity by creating *timeline fragments*. These are pre-computed temporal joins so that users can easily and accurately retrieve the entity's set of attributes for any point-in-time. This is done by solving the overlapping window problem in advance and storing the results in mapping tables.

An account may have attributes stored across multiple tables. For example:

- *Bank.AccountTypeHistory* — tracks changes in account type
- *Bank.AccountLevelHistory* — tracks upgrades or downgrades in account level
- *Bank.AccountStatusHistory* — tracks status changes such as activation, suspension, or closure

Each of these tables is a type II table with [Start date], [End date], and a surrogate key. The goal is to create a *Bank.AccountTimeline* table that expresses the valid combination of attributes for each period.

Example structure of *Bank.AccountTimeline*:

Account ID	Account type SK	Account level SK	Account status SK	Start date	End date
12345	1	1	1	2022-01-01	2023-06-30
12345	1	2	1	2023-07-01	2024-12-31

This allows users to retrieve the full set of attributes for any account at any point in time by joining on the surrogate keys.

[INSERT EXAMPLE OF CODE]

Note that the code is written in a way to bear similarity with normal joins for immutable entities – removing the temporal conditions would give what a standard join looks like.

Third pass — Compute end-of-period tables

The final pass produces **end-of-period tables**—snapshots of the account at regular reporting intervals. These are created by joining the timeline to a calendar table and selecting the row that represents the account's state at the end of each period (e.g. month-end).

For example, joining *Bank.AccountTimeline* on the validity period to a distinct list of end-of-month dates returns *Bank.AccountEndOfMonth*, a ready-to-consume table that captures the correct attributes for each account at the last day of each month.

Building the pipeline to track behaviour

The first pass and second pass apply the same techniques from entity processing onto behaviour records—such as audit results or records of an entity's actions in business processes. These steps filter and structure the raw behavioural data into clean fragments, then compute reusable blocks of information that reflect business interest.

The third pass is where the data engineer infers performance from behaviour. This is still the reduce step, as it involves aggregating detailed behavioural information into insights about the entity. The inference can be simple—such as identifying entities that failed an audit for the first time, or those that have failed three consecutive audits. It can also be complex, requiring statistical techniques such as time series analysis to detect changing trends. At this point, the work begins to transition to data science. Regardless of the technique, the data engineer can express the concept of performance in the data pipeline, reinforcing the business viewpoint and allowing the logic of performance—whether good or bad—to evolve with increasing sophistication.

Following the previous example, *Bank.Transaction* holds records of an entity's actions - the withdrawal and deposits of an account. It is at the grain of [Transaction ID], which is much finer than [Account ID]. To track behaviour of the bank account, the data engineer can aggregate by [Account ID] to produce *Bank.AccountValue*, described by *Bank.RefAccountValue*. Example columns in *Bank.RefAccountValue* would be [Is a millionaire] or [Is a new millionaire in the last 3 months]. These columns support identification of business opportunities.

Formatted: Font: Italic

At an advanced level, behaviours can be converted into type II attributes, effectively transforming behaviour tracking into attribute tracking. For example, if an entity is audited in a business process, the data engineer can define a binary state such as [Has suspicious transactions in the past 3 months] in a type II fragment table *Bank.AccountAuditHistory*. This becomes an attribute with a validity period, allowing

Formatted: Font: Italic

the entity to enter and exit the state over time, which then can also be tracked in the *Bank.AccountTimeline* table.

s 22(1)(a)(ii)

Formatted: Font: Italic

Common problems

Compared with entity processing, entity tracking has additional difficulties.

Hidden tracking problems. It is not always obvious that the business is dealing with an entity tracking problem. Sometimes, tracking is hidden within what appears to be a processing task. For example, a health database may record patient cases involving various types of heart disease. At first glance, this looks like an entity processing problem focused on patient records. Upon closer analysis, the real business interest may lie in tracking the behaviour of heart diseases themselves—how they mutate, spread, and evolve in the population. This would require creating a register table for heart disease types and tracking their macro-level trends. In this sense, the entity tracking of heart diseases is embedded within the entity processing of patient records.

Unregistered entity. A lack of entity registration undermines the clarity of identity. For example, in an online shop with guest checkouts, the absence of registration makes it difficult to track individual shoppers. Similarly, a government department monitoring imported goods from overseas suppliers may struggle to track them if it cannot impose a registration process. In such cases, the entity's identity must be inferred through entity resolution techniques. Broadly speaking, registration *defines* identity before the data record is created; resolution *discovers* identity after the fact.

Undefined memory. Behaviour informs performance. Memory of behaviour informs the view of performance. But it may be unclear how much of the behaviour history is relevant for analysis. Older data may no longer reflect current behaviour. The length of history that contributes to performance is the entity's *memory*. In rare cases, this is explicitly defined—for example, traffic infringements may only be considered relevant for five years. More often, the business has not defined a memory window, making it difficult to determine which historical data should inform current analysis.

Gaps in ~~C~~change history. Tracking mutable entities requires reliable records of historical changes. This can be difficult if the source system does not track changes natively. In such cases, history may need to be reconstructed from audit tables or inferred from surrounding context.

Multiple timelines. Confusion often arises when there is a lag between when a change occurs and when it is recorded. In simple cases, the date of registration aligns with the start of the attribute's validity. In more complex scenarios—such as government registration of social security claimants—back-dating can introduce significant gaps between the business event and its recorded timestamp.

There are three common types of datetime columns:

- **Architectural** — system-generated timestamps, such as those from database triggers. These are mechanically reliable but carry little business meaning.
- **Application** — timestamps defined by the application logic, such as [Update datetime]. These are more meaningful but prone to inconsistency if the application fails to update them correctly or if bypassed by direct database interactions.
- **Business** — timestamps tied to the actual business event, such as [Sales order date]. These are the most semantically relevant but also the most error-prone and often require reconciliation.

The data engineer must engage early with stakeholders to understand the strengths and weaknesses of each type. Clear naming conventions and consistent usage are essential to avoid confusion and promote expressiveness.

Temporal representation. Since attributes can change, the data engineer must decide how to represent the entity at a given point in time. There are three common approaches:

- **Type I** — show the latest available attribute value. The entity appears as a single row with the most recent data.
- **Type II** — show the attribute value as it was at a specific point in time. The entity appears in multiple rows, each reflecting a different historical state.
- **Dynamic type I** — show the latest value as defined by the user's context (e.g., time filter). The entity appears as a single row, but the value changes depending on the selected timeframe.

The choice depends on the analytical scenario. The same attribute may require different representations depending on the business question. Fragment modelling and timeline tables are well-suited to supporting this flexibility.

Computational performance. End-of-period reporting can introduce performance bottlenecks. For example, a self-service model in Power BI that presents entity-level data for end-of-day will consume vastly more resources than one that reports end-of-month snapshots. The difference can be thirtyfold.

Many of these challenges require detailed handling rather than employing ready-made approach. For example, it will take business engagement to determine which of the various dates are appropriate to use for tracking. It would also take business engagement to determine how much of the memory should inform the entity's performance.

Reference data

In the chapters *Entity processing* and *Entity tracking*, we saw the effectiveness of reference data as a tool for expressiveness. Reference tables describe business processes succinctly, provide a place to add descriptive or analytical columns, and can be annotated with metadata to support business meaning.

Beyond expressiveness, reference data is also the cornerstone of integration. This chapter outlines the fundamentals of reference data when applied to a single system. These foundations support the integration of information across multiple systems, which is the focus of the next chapter, *Conforming systems*.

A point of reference

A reference table that applies to multiple business processes becomes a shared point of reference. In dimensional modelling, this is called a conformed reference table. The calendar is a basic example. The list of countries is another. The conformed reference table is also known as the golden record.

s 22(1)(a)(ii)

In rare cases, the mapping of system-specific records to the golden record is straightforward. For example, most processes can map directly to the calendar table using a date column. In more complex cases, systems may maintain their own codes. A business system may have its own country codes. These require the use of mapping tables.

Mapping tables are narrow tables that translate system-specific codes to the golden reference. These are often materialised in advance and stored as standalone tables. In simpler cases, they can be calculated at runtime. For example, if most country codes align but adjustments are needed for the “unknown country” case, a full mapping table may not be necessary.

Applying reference tables

It can be tempting to select a source system table and treat it as the golden reference, asking all other systems to conform to it. This should be avoided. The golden record is treated as physically and conceptually distinct from any source system. This preserves flexibility and supports future changes.

When applying reference tables, the data engineer should proceed in three distinct steps:

1. Model the individual system in isolation without reference data
2. Build reference tables
3. Apply the reference tables for presentation

We illustrate these steps using the example of applying a country reference table to the Cake.Sales business process.

Step 1 - Model individual system

Begin with system-specific tables and build the pipeline as described in earlier chapters. This produces tables such as *Cake.Sales* and *Cake.RefSales*. If the source system includes its own country reference, this would be *Cake.RefCountry*.

[ER SCREENSHOT EXAMPLE]

Step 2 - Build reference tables

Next, construct the golden reference table *Location.RefCountry* and the mapping table *Cake.CountryMap*. The mapping table translates country codes from *Cake.RefCountry* to *Location.RefCountry*.

[SCREENSHOT ER DIAGRAM]

If multiple systems need to map to the golden country list, a single *Location.CountryMap* can be created by taking a union of all records. However, this introduces polymorphic keys. A polymorphic column is one whose meaning depends on the value of another column. A polymorphic key is one whose primary table reference switches depending on the value of another column. While easy to engineer, it is syntactically incorrect from an entity–relationship diagram perspective

[SCREENSHOT DIAGRAM and table].

Step 3 – Apply during presentation

During presentation—typically through a view or dimensional model—a lookup from *Cake.Sales* to *Cake.CountryMap* converts system-specific codes to the golden reference.

[SCREENSHOT SQL]

This process maintains a clear separation between modelling the local system and building reference tables. First, this separation allows the pipeline to proceed while the mapping work catches up. Mapping country codes often requires attention to detail and cannot be rushed early in a project. Second, the separation aligns with fragment modelling. Components such as *CountryMap* and *RefCountry* can be swapped out later if needed.

Separating the build and application of reference tables also provides flexibility in when reference data is introduced as a dependency. Deferring this step is often preferable. Reference data has an amplification effect—changes to a single row can cause widespread downstream updates. For example, if *Cake.Sales* looks up *RefCountry* early in the pipeline and the country code for Australia is lost due to a load error, all Australian rows may be affected. This is a form of instability. The chapter *Fault tolerance* explores this in depth.

The full process creates a minimum total of four tables – *Sales, RefSalesCountry, SalesCountryMap, RefCountry*.

[ER DIAGRAM]

These three steps and the resulting table structure support agility in development, flexibility in components, and stability in the pipeline. These considerations are critical in large, complex systems. In smaller systems, a more lightweight approach may be appropriate.

One option is to skip *RefSalesCountry* if the mapping to the golden reference can be created directly. This reduces the model to *Sales*, *SalesCountryMap*, and *RefCountry*, but places more delivery pressure on having both reference tables ready.

[ER DIAGRAM]

Another option is to skip *SalesCountryMap* if the mapping can be expressed in code. Ideally, this is done using a temporary table rather than an inline conditional. A temporary table preserves the pattern of *SalesCountryMap* without materialising it. This leaves only *Sales* and *RefCountry*.

[SCREENSHOT code, ER diagram].

The decision to simplify should be left to the data engineer, based on the factors above and those discussed in later chapters.

Applying reference data in this way sets the foundation for integrating information across multiple systems in the warehouse.

Conforming systems

One of the highest-value contributions of a data engineer is the ability to integrate information across systems. Yet new engineers often fall into two traps: forcing a union of tables that don't naturally fit, or performing large joins that result in ambiguous grain and duplicated data. Both lead to unclear meaning and *models that are hard to maintain*.

The insight of Kimball's dimensional modelling is that integration can be achieved through ***conformed dimensions***—*reference tables shared across business processes. These allow disparate systems to speak the same language, even when their purposes or implementation differ*.

There are two major approaches:

- **Vertical integration.** This applies when the same type of entity is spread across multiple systems or tables. This is most common when a business process receives a new digital system to replace an old one while keeping much of the functionalities the same.
- **Horizontal integration.** This applies when there are different types of entities, but they have some common denominator for comparison. An example would be a company that has different sub-businesses such as sales and restaurants, but their profits would be comparable on a regional basis.

Vertical Integration

Vertical integration is appropriate when a business entity is recorded across multiple systems. This is because the same different systems are required for different parts of the organisation for different objectives. Another common scenario is when a new digital system replaces an older one and the goal is to integrate information from both in a way that preserves historical continuity.

Take the example of a cake company using two systems, *CakeV1* and *CakeV2*, to record sales. The newer system adds fields and improves logic, but the core attributes remain similar. Integration proceeds in three steps:

1. Model the two systems individually
2. Build reference tables
3. Integrate the transaction tables as union of transaction tables

Step 1 - Model systems individually

A common mistake is to integrate too early. Instead, model each system independently using the principles from earlier chapters. This ensures clarity and prepares each for integration.

Suppose, in the case of integrating *CakeV1* and *CakeV2*, they both have sales, status but V2 has a new attribute for marketing campaign, the tables would be:

- *CakeV1.Sales, CakeV1.RefSales, CakeV1.RefStatus*
- *CakeV2.Sales, CakeV2.RefSales, CakeV2.RefStatus, CakeV2.RefCampaign*

[ER DIAGRAM]

Step 2 - Build reference tables

The second step is to build conformed reference tables that provide the interface for joining the old and new system. The conformed reference represents the unified business meaning across both systems.

In the example of *Cake*, just as the individual systems required *RefSales, RefStatus* and *RefCampagin*, so we need to create the golden reference for these tables – *Cake.RefSales, Cake.RefStatus, Cake.RefCampagin*.

The golden tables can be created in a variety of ways. They can be created manually or derived from the system data. The importance is the express semantic equivalence where both old and new system for references that point to the same real world business concept.

For instance, suppose:

- *CakeV1.RefStatus* uses codes “O”, “C”, “A” for Open, Completed, and Abandoned.
- *CakeV2.RefStatus* uses “OP”, “CO”, “WD”, and “RF” for Open, Completed, Withdrawn, and Refunded.

We might hypothesise that:

- “O” and “OP” both mean Open
- “C” and “CO” both mean Completed
- “A” and “WD” both mean Withdrawn
- “RF” introduces a new concept: Refunded

This leads to a proposed golden reference: “Open”, “Completed”, “Withdrawn”, “Refunded”.

But this is only a hypothesis. It must be validated with the business. Even subtle differences—such as how “Open” is defined in each system—can undermine the integration. Likewise, the business may prefer to collapse “Withdrawn” and “Refunded” into a single status like “Completed without sale”. The final structure must reflect business semantics, not just technical similarity. In this validation, the data

s 22(1)(a)(ii)

engineer should consult with business subject matter experts but also take a critical lens. Business stakeholders may not be familiar with this nuance in complexity and could be mistaken, especially when it comes to older processes.

In addition to aligning codes, the golden reference tables should:

- Include default rows for unknown values
- Add analytical columns (e.g., [Is finished sale]) to support downstream use
- Be documented with metadata to support clarity and reuse

Finally, build **mapping tables**—such as *Cake.StatusMap*, *Cake.CampaignMap*—to translate system-specific codes to the golden reference. These mappings encapsulate the logic and allow for future refinement.

Step 3 - Integrate the transactions

The final step is to integrate the transaction records. It is at this step—and not earlier—that the union of tables is applied to produce an integrated view.

Following the earlier example, we create *Cake.Sales* as a union of *CakeV1.Sales* and *CakeV2.Sales*. If there are columns that exist only in one system, they should be populated with default values in the other, including default foreign key values to reference tables. If there are too many such columns, this may be a sign that vertical integration is inappropriate and a different modelling approach should be considered.

During the union, apply the reference data mapping tables to translate system-specific codes to the conformed ones. For example, use *Cake.StatusMap* to translate the status codes of *CakeV1.Sales* and *CakeV2.Sales* to the common *Cake.RefStatus* values. This step completes the task of conforming both systems to a shared business standard.

[ER DIAGRAM, CODE]

The final structure includes:

- *Cake.Sales* — the unified transaction table
- *Cake.RefSales* — the conformed reference for sales
- *Cake.RefStatus* — the conformed status reference
- *Cake.RefCampaign* — the conformed campaign reference

[ER DIAGRAM]

The separation into three steps—modelling, reference building, and integration—has the same advantages as the three-step approach for applying reference data. It allows each system to be developed independently, supports incremental delivery, and makes it easier to refactor or extend the model later. It also allows a third or fourth system to be integrated using the same pattern.

The above sketch is intentionally over-engineered to illustrate the concept. In practice, simplifications may be appropriate:

- If the concept exists only in CakeV2, such as campaign, then only Cake.RefCampaign is needed.
- If the old and new systems have non-conflicting status codes—i.e., each system has codes the other does not, but no overlap in meaning—then it is possible to directly union the reference tables into a single Cake.RefStatus without requiring mapping tables.
- If differences between systems are minor and easily translatable (e.g., “O” to “OP”), the mapping logic may be applied directly in code, and some mapping tables can be skipped.

Ultimately, the decision to simplify should be left to the data engineer, guided by expressiveness, fragment modelling, and need to build a stable pipeline.

Horizontal Integration

Horizontal integration applies when the entities differ conceptually but share enough commonality to make comparison valuable. The challenge is not technical—it’s recognising when this pattern is appropriate.

Consider a cake company that both produces and sells cakes. The core tables are *Cake.Production* and *Cake.Sales*. These entities are distinct: one records manufacturing, the other records transactions. However, they share common attributes such as region, time, cost, and staff—making comparison meaningful.

New engineers may mistakenly force these into a single abstract table like *Cake.Event* by taking a union of the two. This creates a model that is difficult to understand and prone to error. A better approach is to keep them distinct and rely on shared reference data to support comparison.

For example, both *Cake.Production* and *Cake.Sales* may use:

- *Cake.RefCalendar* to align by date
- *Cake.RefRegion* to align by geography

This allows comparisons such as:

- Cost and staff count by region and month
- Production volume versus sales performance

These comparisons can be implemented using standard joins and aggregations:

[Example....

- ER diagram of calendar, sales, production ...

- SQL: group by date distinct count staff, sum cost, join from calendar.....
- Example Sales region with values North, East, South, West
- SQL: group by date and region, left join from (calendar cross region)]

This approach integrates the data cleanly without distorting the grain of either table. It also fits naturally into a dimensional model, which is covered in the next *Presenting insights*.

Finer-grained integration

Sometimes, business interest demands integration at a finer level than what conformed reference tables offer. For example, the company may want to understand how specific production batches relate to sales. However, the data may not capture this link directly.

Suppose the business proposes a rule: cakes produced in a region are sold in the same region, in the same month, in the order they were produced. This logic is fuzzy because there may be multiple production batches per region per month.

New engineers may attempt to enforce this link by adding [Production ID] to *Cake.Sales*, or [Sales ID] to *Cake.Production*. Both approaches are problematic because they change the grain of the table in a way that is difficult to understand.

The correct approach is to create a **bridge table**—*Cake.SalesProductionMap*—that encapsulates the logic as a standalone fragment. This table contains only two columns:

- [Sales ID]
- [Production ID]

[ER diagram of Cake.Sales, Cake.Production, Cake.SalesProductionMap where the map table has just two columns]

This preserves the integrity of both source tables and allows the mapping logic to evolve independently. The code for *Cake.SalesProductionMap* can be as complex as needed and refined based on business feedback. More examples of these are quoted in the chapter *Meaningful fragments*.

Whether it is the vertical integration or the horizontal integration approach, the key principle is to prioritise expressiveness and good programming. The technical convenience of the data engineer is not sufficient to outweigh these.

Storytelling

Many refer to the principle of garbage-in, garbage-out. This principle is well known, but the challenge to the data engineer goes further. It is not only about avoiding poor inputs. It is about the need to proactively add information by reorganising the data or by annotating it with context to create a new, meaningful way of seeing the data that resonates with business decision-making. So far, we have focused on achieving this through the creation of expressive entities, ensuring that each object in the model corresponds clearly to business interest. In this chapter, we explore three additional approaches: defining the concept of "good" and "bad" entities, trading details for insights, and the use of storytelling dimensions.

Good and bad entities

Business often sees the world through the lens of "good" and "bad." A trade that makes a profit is good; a trade that makes a loss is bad. In government, the distinction is often between compliance and non-compliance. In some cases, only the concept of "bad" exists, as in the case of fraudulent transactions. Yet many digital systems are built mainly to support workflows and lack a nuanced view of good and bad, and thus the concept is missing from the data source.

One of the most effective ways for a data engineer to add value is to apply this lens to business entities. Sometimes the definition is obvious to the business user. Other times, it is latent in the way stakeholders talk about the data. The data engineer can proactively surface this latent definition by listening carefully during discussions and asking the right questions.

Once defined, the concept can be expressed through a reference table and a new fragment. As an overly simplified example, a *Cake.Production* table can be the basis for a *Cake.ProductionFailure* table and a *Cake.RefProductionFailure* table. The reference table might contain a binary column such as [Production failure outcome] with values "Production failure" and "No production failure." It can also be more nuanced and includes [Failure reason]. The fragment table itself would contain just two columns: [Production ID] and [Production failure ID], pointing to the reference table. These two tables alone add powerful meaning for business insight, and beginning to tell a story of the entity.

[EXAMPLE DIAGRAMS]

The definition of 'good' and 'bad' can be a significant point of debate in a complex organisation with different business viewpoints. In these situations, it can be extremely challenging to arrive at a consensus. Many organisations fail to do so, crippling their ability to see issues consistently. In this case, it is often the creativity and technical expertise of the data engineer that can broker between parties by showing the way

forward. This is one of the many examples where the data engineer is an influential advisor in an organisation.

Trading details for insights

As in many fields of endeavour, less is often more. In data engineering, this principle applies directly for surfacing insights.

Take *Cake.Sales* as an example. The business may be trying to identify trends in the market by geography. At the level of individual cities, the data may appear noisy or inconclusive. But when aggregated to the level of region or country, a clear pattern may emerge. Or perhaps we believe that date is a factor in sales. Yet tracking by individual date may show no obvious trend. Instead, aggregating to two-week periods around holidays may reveal a strong seasonal effect. In both cases, a reduction in detail leads to greater predictability or insight.

This is a common phenomenon in data science known as **feature aggregation**—the process of summarising detailed data into higher-level features that improve model performance and reduce noise.

During development, business stakeholders will often ask for the n-th detail. This is understandable, but it can obscure an essential perspective. The problem becomes acute in fast-paced projects where teams tick off requirements and move on. The data engineer must proactively compensate for this tendency. Aggregation should be built into the work plan from the outset.

This is not a compromise, but a necessary step to insight. Aggregation is the act of stepping back to see the forest rather than the trees. When done well, it allows the data engineer to surface insights that are not visible at the level of raw granularity.

The data engineer has three common ways of trading details for insights: creating categories, pivoting combinations, and highlighting special cases.

Creating categories

Creating categories is the simplest way to surface insight that is hidden by noise. For example, aggregating by region rather than city, using age bands rather than individual ages, or grouping days into seasons can reveal patterns that are otherwise obscured.

A special case of this is the use of binary columns. Suppose a help desk workflow has cases in *Helpdesk.Case* and another *Helpdesk.Escalation* table to record an escalation event in a column [Escalation tier] with values from “Tier 1” to “Tier 4”. Creating a new column [Is escalated], defined as false for Tier 1 and true for Tier 2 and above, can be immediately useful for business users seeking to understand escalation behaviour.

A further special case of binary columns is to link categories to “good” and “bad.” For example, an inspection is a common business process. This can be an inspection of

goods, a person, or of a site. The system records inspection results selected from a finite list. For example, *Cake.InspectionResult* can be a table of cake inspections, whose list of inspection values are in *Cake.RefInspectionResult* in a column [Inspection result]. The [Inspection result] may have multiple values such as “Excellent cake”, “Good cake”, “Spoiled cake” and “Not tasty.” An obvious category is to create a column [Is bad cake] that groups the four values into two. This grouping looks straight forward but has a tremendous influence downstream in designing reports, exposing filters, and designing measures.

Pivoting combinations

Another approach is to pivot options in the entity’s detail rows back to the entity grain. Continuing with the help desk workflow example with escalation tiers having values “Tier 1” to “Tier 4”. This detail-level data can be pivoted into a table with four columns—[Tier 1], [Tier 2], [Tier 3], [Tier 4]—to represent whether each tier was reached. Once pivoted, the data engineer can summarise the escalation path in a variety of ways. For example, a column [Escalation path] might contain values such as “Not escalated” (stayed at Tier 1), “Progressively escalated” (started at Tier 1 and worked up), or “Direct escalation” (jumped straight to Tier 3). This column communicates important insight for decision makers trying to understand operational patterns which are otherwise impossible to see with a deluge of tier values.

Highlighting special cases

A third approach is to highlight special cases. For example, if there are multiple results, the data engineer may choose to surface the “worst result,” “first result,” “final result,” or “best result.”

Continuing the help desk example, this could be expressed as a column [Highest escalation] with values such as “Tier 1,” “Tier 2,” and so on. This provides a concise summary of the escalation journey. This is much preferable to displaying a list of escalation tiers to the user and expecting them to filter through.

A common scenario is the header-detail structure where the header is the entity, and the detail is the result. For example, in *Cake.InspectionResult*, a single cake may have multiple inspection criteria, and thus multiple inspection results. A simple, yet powerful, way of storytelling is to compute the worst of the inspection result. If, as in the previous example, the data engineer has created a binary column [Is bad cake] to categorise the result of a single inspection, then the computation would simply be the the max of [Is bad cake]. In other words, starting from *Cake.InspectionResult* and looking up *Cake.RefInspectionResult*[Is bad cake], we can take the max of [Is bad cake] to arrive at *Cake.CakeResult* which defines the result at the cake level.

Summary of trading details for insights

Trading details for insights happens by zooming out. Quite often this is done by aggregating information back to the entity of interest grain. The third pass stage – reduce - in entity processing and tracking is designed specifically to cater for this step in the pipeline.

One useful effect of aggregation is to eliminate problems of double-counting at the grain of business entity. For example, in the help desk scenario, the pivot of options mean that, even though a case may have multiple levels of escalation, the presentation of result can be done at the case level. The users can then add or filter numbers without the risk of double-counting.

In each of these approaches, the expression is best placed in a reference table rather than embedded in the transaction table itself. The data engineer will need to create these reference tables from scratch as they would not be built into the source system.

These three approaches – creating categories, pivoting for combinations, and highlighting special cases – could be combined as one powerful reference data that tells the story of the entity. This reference data is the storytelling dimension.

Storytelling dimension

The concept of dimension will be explored further in the next chapter. For now, it is helpful to introduce a specific type: the storytelling dimension.

The mindset of a storytelling dimension is straightforward. The aim is to tell the overall story of the entity. This means stepping back from what is written in the database and asking what kind of journey the entity undertook through the business processes. The emphasis is on the business view of the situation. The story is then expressed through a reference table rather than in the transaction table itself. This reference table is the storytelling dimension.

There are no fixed rules for creating a storytelling dimension. Trading details for insights is often necessary because the story is often buried in the details, which needs to be traded to surface the story. What matters is that the result communicates a meaningful story about the entity and can be expressed cleanly in a reference table. If both are true, then the dimension qualifies as a storytelling dimension.

Creating a storytelling dimension follows the standard steps for applying reference data: first build the reference table, then map the entity of interest to the appropriate record. The difference with other reference tables lies in the mindset and creativity.

There are three basic steps:

- Build the reference table
- Map to the facts

- Visual check

Step 1 – build the reference table

The storytelling dimension begins with a reference table that captures the **entity's journey through business processes**. Each row represents a distinct path the entity might take—its decisions, escalations, outcomes, or transitions.

The table should be small—no more than 10,000 rows—with each row describing a specific journey. These journeys should categories. These numbers are rule-of-thumb: humans can only comprehend a limited number of paths, and summaries with more than seven categories tend to lose clarity.

Compared to other reference tables, this is the place for creativity. The focus is on **how the entity moved through the business**, not just what attributes it had. The goal is to make the data model resonate with decision-makers by reflecting the real-world flow of events.

Returning to the helpdesk example, a simple storytelling dimension might be *Helpdesk.RefEscalation* with a column [Highest escalation] that labels each case by its most severe escalation tier. This already improves clarity by summarising escalation behaviour without displaying all the detail rows. It also enables bar charts and time series that avoid double-counting.

This can be extended. Suppose *Helpdesk.RefEscalation* includes a column [Escalation path], derived from pivoting [Tier 1] to [Tier 4]. This column might contain values such as:

1. "Not escalated" (easy and not urgent)
2. "Progressively escalated" (not easy but not urgent)
3. "Directly escalated" (urgent cases requiring immediate attention)

These values reflect **how the case moved through the escalation process**, not only how it ended in the highest tier.

The column [Escalation path] can be further enriched when combined with [Highest escalation]. In addition, adding a column [Is closed] is necessary to tell the complete story for all the cases in the business system. Thus, the full combination of [Is closed], [Escalation path] and [Highest escalation] can be used to create [Escalation summary] with values:

1. Closed, directly escalated to Tier 4
2. Closed, directly escalated to Tier 3
3. Closed after progressively escalation

4. Closed without escalated
5. Case still open

These summaries often imply a ranking. Tier 4 is more severe than Tier 3, direct escalation is more urgent than progressive escalation. A [Display order] column should be included to support this ranking in visualisations and filters.

[SCREESNHOT]

In naming, storytelling dimensions should clearly reflect the topic. Avoid generic names like “summary.” Prefer names like *Helpdesk.RefEscalation* or *Helpdesk.RefCaseEscalation* rather than *Helpdesk.RefSummary*.

The reference table should be heavily annotated with useful business descriptions. Each column should describe the business rule it represents. For example, the [Escalation path] column should include a description such as:
“The way a case is escalated in the helpdesk, either not escalated (stays at Tier 1), progressively escalated (starts at Tier 1 and moved up), or directly escalated (starts at Tier 3 or 4). This applies to open and closed cases.”

These descriptions should be written in a way that is close to pseudo-code for business understanding. They help bridge the gap between technical implementation and business interpretation, making the model easier to understand and maintain.

In dimensional modelling, this type of reference table is sometimes called a “junk dimension” or “transaction profile dimension.”

Step 2 - Map to facts

Designing the reference table is the bulk of the work. Once complete, the data engineer only needs to compute the information that matches the business rules described in the reference table’s metadata and column descriptions.

Since the storytelling dimension describes the entity of interest’s journey through business processes, the calculation should be done at the grain of the entity of interest. Most often, this requires aggregating finer-level details, and the computation is performed in the third pass — the reduce stage of the data pipeline.

The calculation can be added to an existing aggregation table if appropriate or implemented as a new fragment. Continuing the helpdesk example, this would result in a *Helpdesk.CaseEscalation* table paired with *Helpdesk.RefCaseEscalation*.

If the data engineer began by clearly defining the reference table and annotating it with metadata that articulates the logic, the calculation will be straightforward. The metadata acts as a blueprint for the transformation, helping to implement with clarity.

Step 3 – Visual check

Visual check is the idea that the data engineer should visually review the outcome of the pipeline, for example, in a bar chart. This should be a constant habit to ensure that implementation matches intent and that no accidental mistakes have crept in.

Visual check is especially important in storytelling dimensions. Crafting a storytelling dimension is a creative act. It often takes multiple iterations to arrive at "the real story." Seeing the outcomes visually through charts, summaries, or dashboards can influence and refine the story being told.

Returning to the helpdesk example, a hypothesis might be to create a *Helpdesk.RefCaseEscalation* column [Escalation path] with values such as "Not escalated," "Progressively escalated," and "Directly escalated." However, a quick bar chart may reveal that there are almost no "Directly escalated" records, and these cases occur only due to data entry errors. Alternatively, the chart may show a high number of "Directly escalated" cases, suggesting a deeper operational issue or a more nuanced story worth exploring.

Arriving at the right story is only possible through visual checks. These checks help validate assumptions, uncover unexpected patterns, and guide the refinement of the storytelling dimension.

Additional comments

In complex cases, more than one storytelling dimension may be necessary. For example, *Helpdesk.RefCaseEscalation* may tell the story of the work to escalate, but another *Helpdesk.RefCaseSla* may need to tell the story of whether the case was resolved within service level agreement with a [Is within SLA] column. Combining escalation path and SLA outcomes will tell a rich and powerful story.

Storytelling dimensions are one of the most powerful tools in a data engineer's toolkit. They allow the engineer to take a fresh look at the raw data and prioritise the perspective of business interest by capturing the journey of an entity in a way that aligns with how the business makes decisions. When done well, they become the most effective bridge between and business understanding. In the next chapter, we will see how these dimensions come to life in a self-service model, where they act as intuitive entry points for users to explore, filter, and interpret the data.

Meaningful fragments

New engineers often build giant tables to answer queries. This feels convenient and intuitive—everything in one place, easier to visualise. However, this approach has serious drawbacks, which we will continue to explore throughout the text.

Meaningful fragments is one of the most important concepts in data engineering.

It is the idea that, instead of creating wide tables with many attributes, the developer creates **narrow tables with specific, self-contained meanings**. These fragments are designed to:

1. Represent expressive content
2. Package information in reusable blocks
3. Isolate complex computation
4. Enable change detection specific to those columns
5. Reduce dependencies across the pipeline

The general approach is to extract data from the source as-is, keeping only the necessary columns and rows, and then compute everything else in meaningful fragments.

Restaurant analogy

In a restaurant kitchen, meaningful fragments are like prepared plates of chopped vegetables, cured sauces, or marinated proteins. They are ready to be used as needed. The raw data are like whole potatoes or unwashed greens.

- The first part of processing is to peel off the skins and discard the junk
- The second part is to prepare mini plates, which in kitchen terminology is the step of *mise en place*
- The third part is the actual service—combining ingredients to cook and present the dishes on time

The second part, *mise en place*, creates mini plates, which are the equivalent of meaningful fragments. As one notable chef said, “When the mise-en-place is done well, the service is easy.” In the same way, with the correct fragments, surfacing the insights for use is easy.

Symptoms of anti-pattern

When meaningful fragments are not used effectively, several symptoms are obvious:

- The same logic reoccurs in WHERE clauses or join conditions across multiple queries – indicative of the fact that the logic could have been centralised earlier

- Tables become wider and wider as constructed columns are bolted onto fact tables, making them hard to reuse and maintain
- Information in the data model becomes duplicative of other tables, especially reference data, causing confusion for uses
- Users must write excessive joins and bespoke filters just to answer basic questions, rather than just plug-and-play

These symptoms indicate a lack of fragment modelling.

Examples of meaningful fragments

The following list provides illustrative examples of meaningful fragments. These are common patterns, but not exhaustive. Each serves a distinct, common, scenario in modularising the pipeline.

Summaries of details and storytelling. Any reasonably complex business process will generate a significant number of detail rows that must be summarised to be useful. This is the subject of the previous section on *trading details for insight*. The computation for these summaries tends to be complex and should be placed in a separate fragment rather than compounded onto existing tables.

A special case is the storytelling dimension and its related fact. Once computed, the storytelling dimension becomes the primary window through which the business sees the data. It often becomes the default filter in queries. Because it is aggregated at the entity of interest level, it enables elegant queries and avoids the need for users to query subprocesses and then deduplicate results.

Milestone datetimes. Complex business processes often involve multiple milestones. For example, a helpdesk may want to know when a case was received, first actioned, escalated, and completed. A manufacturing company may want to track when products were produced, shipped, and sold.

Entity processing is always interested in measuring the time between milestones. This can be difficult if steps repeat or are inconsistently recorded. The data engineer can add value by precomputing these milestones into a dedicated fragment. A **milestone datetimes** fragment is simply a table at the grain of the entity of interest, with one datetime column per milestone. This makes it trivial to calculate time gaps and generate histograms of time taken.

[INSERT SCREENSHOT of table and histogram]

Milestone tables known as “accumulating snapshots” from dimensional modelling.

Current or primary version. As discussed in the chapter on *Entity processing*, immutable entities often have multiple versions. While the full version history may be

useful for audit or lineage, the business is frequently interested in only the latest valid version that reflects the current state of the entity.

In many systems, this logic is handled in the application tier and is not obvious in the database. Rather than expecting users to reconstruct this logic repeatedly, the data engineer should precompute the latest version in a dedicated fragment.

In some cases, versions may not follow a strict chronological order. For example, multiple submissions of the same entity may exist, and the business may need to designate one as the **primary version**—either for reporting, counting, or representation. This selection may depend on business rules, such as status flags, timestamps, or user roles.

Whether the requirement is for the **current version** or the **primary version**, if the logic is non-trivial, it should be implemented in its own fragment. This avoids compounding complexity in core tables and ensures that the logic is transparent and replaceable.

This is a case where fragment modelling can lead to a loss of expressiveness. Most users do not expect to retrieve a row from one table and then consult a separate fragment to find the current version of that row.

The data engineer can compensate for this by ensuring that the main table is clearly named—for example, including the word “version” in the table name and primary key columns. Additionally, the fragment should include relevant metadata, such as foreign keys, to make the relationship between versions and entities transparent.

This naming and metadata strategy helps preserve expressiveness while maintaining the modularity advantages of fragment modelling.

Timeline and end-of-period. As discussed in *Entity tracking*, when an entity has evolving attributes stored across multiple tables, it becomes difficult for users to retrieve consistent point-in-time views. This can be simplified through:

- **Timeline tables**: sets of keys pointing to Type II attributes for each validity period
- **End-of-period tables**: snapshots of those attributes at the end of each reporting period

These fragments make temporal analysis accessible and reliable.

Mapping. Business processes often do not record relationships at the grain needed for analysis. Sometimes this granularity is impractical. These relationships can be approximated using fuzzy logic in the analytical layer.

These are often many-to-many relationships. Rather than disturbing the grain of core tables, the mapping can be stored in separate fragments—typically two-column tables with foreign keys pointing to each primary table. This is the **mapping table**.

Distribution weights. In some cases, relationships lose information about how entities relate. A similar but distinct case is when measures are not clearly recorded against entities.

For example, a system may record the total time to complete multiple tasks, but not the time per task. The business may have rules to distribute the total time back to individual tasks as weights. This logic can be complex and should be calculated in its own fragment. This is the **distribution weights** table.

In many cases, mapping tables and distribution weights may be combined if their ambiguities stem from the same business phenomenon.

Hubs. A concept from the *Data Vault* method of data warehousing. A **hub table** creates a consistent surrogate key for the same entity appearing across multiple systems.

For example, a company may have multiple subsystems recording staff names, identified by first and last name. A data engineer may need to create a consistent surrogate key to represent this pair across all systems—either for anonymisation or for use as a relationship key in Power BI.

A hub table is a centralised fragment where this pairing and key assignment occurs. It belongs to no single subsystem and should be computed independently.

Conclusion

These are examples of common patterns of fragments. In practice, anything can be a meaningful fragment—it does not need to fall under a predefined category. As long as the information is self-contained, and its creation can be justified by complex computation, dependency management, or business meaning, it qualifies.

The ability to create meaningful fragments also reflects the ability to see the data product in terms of its **minimal informational components**. This “informational view” is a key to arriving at efficient, maintainable designs. It is one of the hallmarks of a mature data engineering mindset.

Presenting insights

The craft of dimensional modelling

Creating information focused on building reusable blocks of information. These are *meaningful fragments*. This part, *Presenting insights*, looks at turning them into consumable products that deliver answers in a way that is easy, intuitive and accurate.

The modern rise of technology has propelled the ways in which data engineers are expected to embed data insights. Machine learning models, embedded analytics, and real-time applications have all areas where a data engineer may need to present insights. However, one method has stood the test of time and is thus the essential skill for a data engineer. It is the discipline of dimensional modelling.

Dimensional modelling, pioneered by Ralph Kimball, remains a go-to approach for communicating business insights. More recently, Microsoft Power BI semantic models have become a popular tool for business intelligence. It is particularly adept at allowing non-technical business users to interact with their data in an intuitive way, thus lowering the barrier to data analysis. A Power BI semantic model is also useful for producing multiple reports. At its core, Power BI is also centred on dimensional models. Hence, a dimensional model in Power BI is a powerful vehicle for surfacing insights, and is the subject of this part of the text.

While building a data pipeline is largely a scientific exercise grounded in computer science mindset of minimal information, creating a dimensional model in Power BI is more of an art. While technically demanding, its central challenge lies in anticipating how users will interact with the product. There is a focus on the model's look and feel, its clarity, and its intuitive alignment with business intent. Dimensional modelling, therefore, is a craft shaped by experience and judgement.

It is not difficult to build an effective dimensional model. If the data engineer has laid a strong foundation with meaningful information fragments that become facts, expressive reference tables ready to be converted into dimensions, and binary flag columns that simplify measure definitions, then building the model is simply a matter of assembling these elements in a way that anticipates user queries.

The real challenge for a data engineer is not *how* to build the dimensional model, but *what* to build. A useful starting point for new data engineers is to become familiar with two guiding questions: *What does a good dimensional model do?* and *What does a good dimensional model look and feel like?*

A good dimensional model

A data engineer can become familiar with what a good dimensional model in Power BI should look like through two aspects: the *expectations* and *indicators*.

Expectations

A good dimensional model in Power BI should:

- Resonate with the business view
- Be intuitive and unambiguous – it should "just work"
- Anticipate questions
- Be both high-level and detailed
- Be performant to use

Resonate with the business view

While the data engineer must always focus on expressiveness, nowhere is this more critical than in the dimensional model. In some cases, technically correct representations may need to give way to what better reflects how the business sees the world. For example, relaxing strict grain consistency may allow a model to align more closely with business understanding, rather than having a proliferation of objects that are technically correct but hard to understand.

Resonance begins with naming. Models, tables, columns, and measures should use language that is immediately meaningful to the business. This may require creating perspectives the business has not yet articulated. Familiarity can often lead business areas to miss insights; the data engineer, working across domains, is well-placed to offer a fresh view.

Metadata plays a key role. Descriptions should speak the business language, clearly state the intent, and for complex cases, include elaboration in near-pseudocode using business terms.

Be intuitive and unambiguous

As a self-service data model, the user is expecting to drag-and-drop to get answers. For this purpose, the model should *just work*. If the user must memorise complex technical filtering rules, or memorise a map of the model relationships to get the correct answer, then the model is not intuitive. If the user has to stop and think, or second-guess the logic of the model, then something is wrong.

An intuitive model means that the user should be able to operate it without a heavy manual. It should behave in a way that matches expectation with no hassle.

A related phenomenon is ambiguity. An ambiguous model means that there are apparently multiple ways of getting the same information, or the same information being open to interpretation. As an example of the former, this could be when a column has been denormalised into multiple fact tables, leaving the user unsure which one to

use for a particular query. As an example of the latter, it could be because a column name such as [Calendar date] is too generic, and it's not clear which business date the column refers to.

An unambiguous model means that the user has only one obvious way of answering a question. The model should either answer it accurately or make it apparent that it cannot.

Anticipate questions

A good model does not merely satisfy the initial requirements. It anticipates the full range of reasonable business questions. This requires the data engineer to think beyond the immediate requirements and consider *all* the questions that the model may need to answer over time. There is no way to hide from this reality. If the question is not answered now, at some point, a business area will come back and ask for the question to be answered later.

In practice, what this means is that either a business process is out of the model, or if it is included, then all information captured by that process should be accessible in a format to meet the expected usage scenario.

Achieving this is much easier than it sounds, and is the subject of the chapter *Anticipating questions*.

Be both high-level and detailed

One of the most powerful features of Power BI is the ability for users to interactively move through the data by cross-filtering and drill-through. This invites a natural behaviour where users start at the high-level view, and if something catches their attention, they drill into the details to see exactly what happened. This is an integral part of how users build trust in the model through examining detailed records.

Equally, users who operate at the detailed level often start at the transactions for the purposes of troubleshooting, but will eventually want to zoom out in search of broader patterns. This zooming out is a sign of trust and maturity.

A data engineer cannot ignore this reality of both satisfying both the high-level and detailed views. While reports and specific queries may have one or the other view, there are many cases where the underlying model must support both.

s 22(1)(a)(ii)

Be performant to use

Most queries should return in under a second. Complex or infrequent queries should complete in under two seconds. Longer runtimes are acceptable only in rare cases.

Performance is not merely a user-experience consideration. It is a symptom of underlying issues to do with information articulation. A slow model often indicates that

key information has not been pre-computed so that the model must compensate with work at query time. Thus, a slow model may be a concern of business expressiveness.

Indicators

The following are *indicators* of a good dimensional model in Power BI. Unlike the expectations, which describe what a model should achieve, indicators reveal whether those qualities are present in practice. They are not all mandatory and occasionally may be violated. However, a good dimensional model should have a strong presence of these indicators. Their absence or frequent violation would hint at a model that is difficult to use.

The indicators can be categorised in aspects:

- Names and metadata
- Dimensions
- Facts and measures
- Relationships

Names and metadata

All names should be business-centric rather than system-centric. It is a common mistake for data engineers to use acronyms or terms that refer to the digital system rather than the business content itself. This should be avoided unless the information to be presented is intentionally about the system itself.

Names should not be technical. Terms such as “dim,” “fact,” or “bridge” should not appear in visible names. These are implementation details and do not reflect business meaning.

Names should be explicit. For example, rather than [Date], it is better to use [Sales date]. It may be necessary to repeat the table name in the column name, such as *Sales*[Date] to *Sales[Sales date]*. This is useful for clarity but also because the column name is the one that appears in the report visual rather than the table name.

Visible names should not repeat across the model. When a user searches a name in the field list, it should return only one result. If a column appears more than once, it presents an immediate ambiguity. If the columns are actually the same but all of them need to be visible to the user, this indicates the model needs to be reworked to resolve duplication of information. If they are different columns but using the same name by accident, then the columns need to be renamed for clarity rather than relying on the table context for resolution.

Dimensions represent information or attributes, while facts the business processes. As such, dimensions should be nouns and business process facts should be verbal nouns, or contain a sense of action. For example, in a business system with three processes, manufacture, order, and shipping may have facts called *Manufacture, Order,* and

Shipping. The dimensions may be called *Product*, *Region*, *Manufacture date*, *Order date*, and *Shipping date*. Thus, the facts are the names of the business processes, while the dimensions are attributes that the business processes capture.

Naming the facts after the business process is harder to do than to say. It requires a laser focus on the business intent, and familiarity with Power BI functionality to design a model with this simplicity.

A common case is the transaction and reference table pair, such as *Sales* and *RefSales*. One approach is to use *Sales* as the fact name and use *Sales description* as the dimension.

Finally, a strong indicator of a good dimensional model is the presence of rich, business-centric descriptions in the hover text of tables, columns, and measures.

Dimensions

In Power BI, dimensions are the window through which the user interacts with the model. A good dimensional model consciously focuses on dimensions as the user's point of access. They are the biggest influence on the model's "look and feel."

On a surface level, dimension table names should be nouns that convey the type of information available to the user. Dimensions should have a reasonable number of columns—approximately 5 to 10. There can be more columns if some of them are simple derivation of the others. For example, text versions of binary flags.

Too few columns in a dimension mean that it carries too narrow a business meaning. For example, a dimension called *Gender* with a single column [Gender] has too few columns. A large number of dimensions with few columns represent a fragmented, rather than a consolidated, view of the business. It may be appropriate to combine *Gender* with a few other dimensions to form a *Demographic* dimension. Too few columns may also point to a lack of helpful attributes. For example, the *Gender* table should at least have a [Gender display order] to sort the values in [Gender] in a useful way.

On the other hand, too many columns in a dimension suggest a grouping of attributes that is too complicated and may overwhelm users. It also reflects a view of the business that is insufficiently nuanced. In this case, the dimension may need to be broken down into a few more manageable dimensions.

A dimensional model often includes two special types of dimensions that indicate a thoughtful design. The first is the ID dimension. It is a table of the primary keys of the business process. This table allows users to retrieve all the detailed transactions for any business entity by looking up a familiar business key. The second is the storytelling dimension. It sits at the opposite end of the spectrum. As explained in the *Storytelling* chapter, the storytelling dimension categorises all business entities into a digestible

number of journeys. This provides the best possible high-level portal for users who take a zoomed-out view.

The presence of both ID dimensions and storytelling dimensions is an indicator that the data engineer has covered the broad range of needs the data engineer is expected to meet. While it is not a guarantee of quality, it is a good rule-of-thumb for initial assessment.

The full set of useful dimensions are in the chapter *A primer of dimensional modelling*.

Facts and measures

Power BI is designed such that the primary way of interacting with the data is through dimensions and measures rather than facts. This is explained in greater depth in *Filtering behaviour*. For now, it suffices to note that an indicator of a good Power BI dimensional model is a *reduced* prominence of the fact tables. In the ideal case, all fact tables are hidden from the user.

A related indicator is the absence of, or at least limited use of, degenerate dimensions in fact tables. A proliferation of degenerate dimensions suggests the model is working against the inherent nature of Power BI to propagate filters. It also indicates that certain business information are left as miscellaneous attributes of the business processes instead of being articulated as standalone properties that deserves a dimension table.

Measures play a more prominent role that corresponds to the diminished role of fact tables. An indicator of a good dimensional model is one that invests heavily in measures. A rough rule-of-thumb is that each fact table, representing a business process, should have at least ten measures. This simply reflects that there are usually at least ten metrics to understand a business process. For instance, a process such as sales would have measures for sales volume, turnaround time, profit, and cost. If a model has implemented the concept of “good” and “bad”, this leads to sub-measures for individual categories. In addition, there will be measures for percentages and other derivatives. This rapidly adds up to more than ten.

If a user can easily grab-and-go to use measures with minimal effort to arrive at desired answers, then it is an indicator of a good model. It shows that the data engineer has thought deeply about the business processes and is able to put the necessary information at the user’s fingertips. Conversely, the of lack ready-to-use measures mean that the data engineer has not sufficiently engaged with the business interest of the user.

When there are many measures in a model, it can become unwieldy. Good measure management places all measures into a single table at the top of the field list. A suitable name for this table is simply *Measure*. Power BI promotes this behaviour. If all

columns in a table are hidden and it contains at least one measure, Power BI places this table at the top. This indicates a model that elevates measures to first-class.

The measures in the measure table should be organised by display folders. By default, folders should be grouped by business process. For example, a business system with the processes *Manufacture*, *Order*, and *Shipping* can have at least three folders: "Manufacture", "Order", and "Shipping."

[SCREENSHOT?]

The prominence of measures, their business-centricity, and their organisation through business processes are strong contributors to the look and feel of a good data model.

Relationships

Relationships in a Power BI model define user interactivity. Following the theme of using dimensions as the main point of interaction, dimensions should filter facts and not the other way around.

An indicator of a good dimensional model is that if a piece of information is known to a business process, then the dimension representing that information should filter the corresponding fact. For example, in a business system with *Manufacture*, *Order*, and *Shipping* (in that order, and recorded at the unit level), the *Product* dimension should filter all three facts because all those processes know about the product. On the other hand, the *Shipping date* should filter only the *Shipping* fact. It should filter neither *Order* nor *Manufacture* because only the shipping process knows the shipping date but not the earlier processes. This does not mean that each business process is filtered only by its own attributes. For example, *all* fact tables should be filtered by the *Manufacture date*, because the shipment tracks back to the order, which tracks back to the manufacture, and therefore all processes know about the manufacture. The criterion is whether the information can be reasonably related to the business process at the time. Relationships are explored in greater depth in *Anticipating questions*.

Since dimensions are the user's access point to the model, the column on the filtered side of the relationship (fact) is always hidden to avoid ambiguity. Instead, the column on the filtering side (dimension) is used.

Conclusion

This chapter focuses on sketching out what a good dimensional model looks like. The purpose is not to provide a definitive checklist that guarantees quality, but to attune the data engineer to the look and feel of a good model in Power BI by putting the engineer in the perspective of the user. While a list of indicators is helpful, the best test is for the data engineer to actively test the model personally from the perspective of the user, trying a range of business questions to ask. This is part of conducting visual checks.

s 22(1)(a)(ii)

This task requires a deep familiarity with the business and a strong curiosity of the subject matter, rather than an attitude of ticking off a set of formal requirements.

The common theme behind the indicators is business centricity and intuitive use. An intuitive model is achieved through explicit implementation of answers that pop out, and does not require the user to interpret or guess. The data engineer achieves this by crafting the dimensions and providing a rich set of measures in the model.

The Zen of Python is a good guide to designing a dimensional model. Its emphasis on explicit over implicit, and on having one—and preferably only one—way of doing something, applies directly to designing a model in Power BI.

s 22(1)(a)(ii)

A primer on dimensional modelling

Dimensional modelling is a well-established discipline. Power BI, by contrast, is a more recent technology that brings a new approach to implementing dimensional models. Its unique features allow for a special kind of dimensional modelling that fully supports the classical approach, but permissive enough to break traditional rules when appropriate.

New starters often swing between two extremes. On one end, they apply Kimball dimensional modelling techniques too literally in Power BI, resulting in a proliferation of fact tables that makes the model difficult to understand. On the other, they abandon dimensional modelling altogether, allowing Power BI's flexibility to take over and creating a model that quickly becomes a relationship nightmare.

What follows is a practical guide to dimensional modelling as seen through the implementation in Power BI. The aim is to help developers build models that are intuitive and robust. This approach does not contradict traditional dimensional modelling but supplements it.

Understanding facts and dimensions

Classical dimensional modelling revolves around two core concepts: facts and dimensions. In a technical view,

- **Facts** are the *transactions*—the events or activities that occur in the business.
- **Dimensions** are the *reference data* —low cardinality tables that provide look-up of contextual information for the transactions.

Fact tables are fast-moving tables. They grow quickly, have high cardinality, and record frequent business activity. Dimension tables are slow-moving tables and usually have a low cardinality.

In contrast to the technical view, a business view sees:

- Facts as **business processes**—these are the verbal nouns and convey action, such as *Manufacture, Order*, or *Shipping*.
- Dimensions as **attributes or information**—these are the nouns and convey objects, such as *Product*, *Region*, or *Customer*.

While most data engineers are familiar with the technical view, the business view is also essential to building a business-centric dimensional model.

In most explanations of Power BI, dimensions are described as filtering facts. This is technically accurate, but it is also helpful to think about it from the users' experience when they interact with the model.

To use an analogy, think of dimensions as *levers or switches*, and facts as *robots or machines*. If we think of filtering as a form of *control*, then:

1. Levers control robots
2. Levers occasionally control other levers
3. Robots occasionally control other robots
4. Robots should never control levers

This analogy helps reinforce a key design principle: interactivity should be driven by dimensions. In the same way that levers are what users control, so the dimensions are the users' interface to the model.

In addition, rather than switches on the robots themselves, they are placed on the controller. In practical terms, this means users should interact with dimensions to drive filtering, not with columns embedded in fact tables (degenerate dimensions).

In this primer, we stick with a simpler view of dimensions. In more advanced scenarios, dimensions can be seen as **buttons** that users click to produce certain **effects**. The fact tables contain the **ingredients** to generate those effects. Abstractly, a dimension may even be a set of tables working together, and they do not even need to filter the fact tables, or any table at all. The only criterion is that it operates as a button. If it is something the user interacts with to create a certain effect, then it can qualify as a dimension.

Elements of a dimensional model

Literature abounds on dimensional modelling. This section does not seek to replace it but reframes the well-understood artefacts of dimensional modelling as a repertoire of building blocks for the data engineer. These blocks allow the data engineer to construct an interactive Power BI model that addresses a range of usage scenarios. The view here is functional, not technical.

As elements of a Power BI dimensional model, there are 3 types of facts and 9 types of dimensions.

Facts:

1. Measurable fact
2. End-of-period fact
3. Annotation fact

Dimensions:

1. Business dimension
2. Role-playing dimension
3. Combination dimension

4. Choices dimension
5. Sankey dimension
6. Histogram dimension
7. Storytelling dimension
8. Transaction or search dimension, a sub-type being the ID dimension
9. Degenerate dimension

Facts

Measurable facts are the basic transactions and are what people usually think of when they think of facts. Examples include inspections, sales, and audit events. What makes measurable facts stand out is that they are designed to work with aggregated measures, such as *SUMX* and *COUNTX*, and their content are presented via these measures. Measurable facts form the backbone of most analytical models and are the primary source of business metrics. They are the default facts for entity processing scenarios.

End-of-period facts capture the state of an entity at the end of each reporting period. An example is the attributes of an employee at the end of each month. They are designed to support end-of-period measures, such as count of employees at the end of the user selected period. End-of-period facts are the default facts for entity tracking scenarios, allowing the model to reflect the evolving state of entities at specific points in time. They can be read in from the end-of-period fragments in the data pipeline.

Annotation facts provide additional, finer grain, details of a transactional such as free-text comments for an inspection or certificate details for a product. The reason for moving the additional details to its own fact, rather than leaving them as a degenerate dimension, is when the additional details have a higher cardinality than the transaction itself. For example, one inspection may have many comments. Including them in the business process fact would either force an unwelcome change in grain, or require the use of string concatenation. When neither option is desirable, the annotation fact is the right solution.

Annotation facts hang off another fact by being in a child relationship with it. The information in an annotation fact is designed for display rather than for aggregation. The typical usage scenario is one where the business user has arrived at a set of transaction records through a dimension and would like to see additional details about that transaction. This is also why they are filtered through another fact, not a dimension.

s 22(1)(a)(ii)

In general, the relationship would be many-to-many. For example, an *Inspection* fact table may have multiple inspection results. On the other hand, one inspection may have multiple comments which are represented by the '*Inspection comments'* annotation fact. These both relate to the inspection itself, the relationship would be many-to-many on [Inspection SK].

Annotations are one of the rare cases where bidirectional filtering and fact-to-fact filtering is advisable. This relationship setup is not problematic because annotation facts sit at the tail end of a filtering path and thus pose limited usability issues. Since they are not usually aggregated, they also pose no performance problems.

[SCREENSHOT]

Often, a measure is still needed to display the additional details in a controlled way. For example, the *SELECTEDVALUE* function can ensure that additional details are only shown when a primary key value, the full key or parts thereof, for a transaction is used.

In linguistics, an adjunct noun is a noun that functions as an adjective to describe another noun, such as *vegetable* in vegetable soup. In this sense, an annotation fact is an adjunct fact.

Dimensions

Business dimensions are the common dimensions that most people think of when thinking of dimensions. Examples include processing statuses, locations, officers, inspection types. They are usually available out-of-the-box from source systems, though they may be supplemented with additional useful columns by the data engineer. These dimensions are necessary for presenting basic information and form the foundation of most models.

Role-playing dimensions are dimensions that have different meanings depending on the facts they filter. Technically, a role-playing dimension can be identified through a primary key that links to multiple different foreign keys. For example, a reporting calendar may filter on three facts — *Manufacture*, *Order*, and *Shipping* — simultaneously. It is visible as a role-playing dimension if the foreign key columns — [Manufacture date], [Order date], [Shipping date] — are different columns. However, if the calendar filters on the same foreign key, for example [Manufacture date] on all three facts via inheritance of information, then the calendar is not a role-playing dimension. In this case, the calendar always means the manufacture date.

Role-playing dimensions are important when comparing multiple measures against the same reference point, such as calendar or staff. The hover text description for a role-playing dimension should clarify the meaning it takes in the different scenarios it can play.

A **combination dimension** is a set of binary columns that describe whether a business entity exhibits one or more similar properties. The presence of multiple flags allows the model to express combinations of these properties.

A simple example is a *ColourFilterCombination* dimension with the columns:

- [Has red filter]
- [Has blue filter]
- [Has yellow filter]

This structure allows the model to represent entities that have any combination of these colours.

Some business entities, especially in complex organisations, may have more than one digital system touchpoint. In this scenario, business users are often interested in which systems the business entity was processed through. A *SystemCombination* dimension with the binary columns [System A], [System B], and [System C] will easily support this scenario.

The combination dimension is particularly useful when a business entity may carry multiple types in its details. For instance, a cargo may contain line items. Some are fresh produce, and some are inanimate goods. A combination dimension would express this as:

- [Has fresh produce]
- [Has inanimate cargo]

From these base flags, the data engineer may derive additional columns such as:

- [Has mixed cargo] — a computed flag indicating the presence of both types.

These derived columns are not part of the combination itself but are logical consequences of it. They are useful for storytelling, filtering, and simplifying measure definitions. The key benefit of this use is that it aggregates detail-level complexity back to the entity level, avoiding grain expansion and preventing double-counting.

Choices dimensions are multi-valued dimensions of low cardinality. They are like combination dimensions but differ in intent. The business is interested in whether the entity has *any* of the attributes, such as in multi-select scenarios. Choices dimensions are particularly useful when the number of possibilities exceeds ten or more. A use case is displaying data quality issues. For example, if a transaction row has multiple problems — missing date field, missing officer name, etc — then a choices dimension allows users to display on transaction rows that have *any* of the issues.

Physically, choice dimensions are created through taking all possible choices of a finite list and then assigning each set of choice with a group number. The relationship is between the fact and the dimension is on this group number.

[SCREENSHOT EXAMPLE, Tea, coffee or both]

Choice dimension takes advantage of Power BI's ease at handling many-to-many relationships. Like the combination dimension, it allows the data engineer to work push information back to the entity of interest level. Without this dimension, joining any information to the choices would require changing the grain of the original information. This can create an unwieldy output. By using the choice dimension, the desired information can propagate conveniently to all business processes without changing the grain of any of the interim output.

s 22(1)(a)(ii)

Choice and combination dimensions are closely related. Combination dimensions are ideal for expressing *and* logic; choices dimensions are ideal for expressing *or* logic. In addition, the data engineer can create a choices dimension by in pivoting a combination dimension and discarding empty values. For example, the *SystemCombination* dimension with columns [System A], [System B], and [System C] can be unpivoted to create a *System* dimension with column [System touch point] with values "System A", "System B" and "System C." Depending on the scenario, the business may be interested in seeing records that involve a combination of systems to process a business entity (combination dimension), or seeing records that require *any* of the system (choice dimension).

[SCREENSHOT]

Thus, a data engineer can always build both in the data pipeline and use one or the other, or both, in the dimensional model as needed.

A **Sankey dimension** is used to represent possible paths through a series of sequential checkpoints. Suppose there is a business process with a list of checkpoints which are sequential, but some are optional. Consider all the possible paths through these checkpoints, and designate them with a number "Path ID." A Sankey dimension is a table with three columns [Path ID], [From checkpoint], and [To checkpoint] where each row is an edge in this path.

For example, a helpdesk may have escalation points ranging from Tier 1 to Tier 3. There are 7 possible pathways depending on whether a case reached any of the checkpoints. The Sankey dimension is a table of all possible transition from tier to tier. A starting row may be necessary to illustrate the entry point for every path. The table would then be:

Path ID	From checkpoint	To checkpoint
1	Start	Tier 1
2	Start	Tier 2
3	Start	Tier 3
4	Start	Tier 1
4	Tier 1	Tier 2
5	Start	Tier 1
5	Tier 1	Tier 3
6	Start	Tier 2
6	Tier 2	Tier 3
7	Start	Tier 1
7	Tier 1	Tier 2
7	Tier 2	Tier 3

This dimension is perfect for creating flow diagrams without tampering with the grain of the fact tables. A case in the help desk must go through one of these seven possible journeys, and thus have an associated [Path ID] value. This case in the fact table can relate to the dimension on [Path ID]. Like the choice dimension, it is a many-to-many dimension.

[SCREENSHOT]

In practice, a Sankey dimension can be created through a combination dimension, and then unpivoting the checkpoint columns, and then calculating a LEAD() to bring in the next checkpoint ordered by the sequence. Computationally, this can become unmanageable if the number of checkpoints grow quickly.

[SCREENSHOT]

Histogram dimensions are used when it is necessary to filter the model by a numeric value. For example, finding the number of helpdesk cases that take more than 3 days to resolve. Measures such as [Total days to process] cannot be used because measures

s 22(1)(a)(ii)

do not work as normal filters in Power BI. Instead, the numeric value needs to be expressed as a dimension.

A histogram dimension is table of values that span a range of numeric in incremental steps. This range can be absolute numbers moving in even steps, such as integers from -100 to 100 in steps of 0.5, or it can be expressed as band of numbers, such as 0 to 10, 11 to 20, ..., 91 to 100. This dimension map to the fact tables on the numeric value and thus translates them to a dimension for interaction. Care must be taken on the boundary steps to ensure correct assignment, especially on the maximum and minimum. An additional display column may be necessary for cases such as “Greater than 100” and “Less than -100”.

As well as allowing the user to filter the model on a numeric value, the dimension is useful for creating histogram visuals – hence the name. This chart is perfect for visualising time-to-process metrics by counting the number of entities against a dimension of days or hours to process.

Storytelling dimensions are covered in detail in the *Storytelling* chapter. They are artificial dimensions that narrate the journey of an entity through business processes. Like combination dimensions, their distinction from a typical junk dimension lies in semantics, not syntax.

A storytelling dimension is created when the data engineer steps back from the raw data and envisions afresh what the business perspective looks like. The engineer then categorises all entities of interest into a finite list of journeys that reflect how those entities move through the business process.

In the context of a self-service model, storytelling dimensions play a powerful role. They act as the first portal into the data, dividing all entities into manageable strata and initiating the process of drillthrough and closer examination.

A close equivalent in traditional modelling is the “transaction profile dimension.”

Transaction dimensions are high-cardinality dimensions where each transaction row is a dimension value — that is, the dimension *is* the transaction itself. Their primary purpose is to allow drillthrough or cross-filtering of a single transaction across multiple business processes within a system. They are important because some high cardinality attributes are useful for comparison across multiple processes but do not easily normalise into standalone dimension due to its high cardinality.

An important special case of a transaction dimension is the **ID dimension**. By design, a transaction dimension must include the primary key of the transaction to identify it. An ID dimension is one where the dimension contains only the primary key and almost nothing else, perhaps it includes a [Version number] column. Such a dimension allows a business user to quickly retrieve any information from the model using a known

business key. ID dimensions play a prominent role in Power BI because of its built-in interactivity. In traditional dimensional modelling, they would be the exception rather than the rule. In Power BI, this becomes the reverse.

A more general view is to treat these as **search dimensions**. A search dimension is a high-cardinality dimension that supports searching business processes through a high-cardinality value. For instance, the free-text description of mail packages may be used as a dimension that allows users to search business processes based on that text. In this case, the search dimension corresponds to the annotation fact. When the business objective is to search through a free-text field, it is best implemented as a search dimension. If the user is not expecting to search using the field, but simply wants to display supplementary information, then it is better implemented as an annotation fact.

Transaction dimensions add significantly to model size and are not ideal for aggregation. This is why their primary purpose is searching and cross-filtering, not aggregation.

Due to Power BI's filtering behaviour, transaction dimensions need to be used in conjunction with a unit-record display measure. This is explained in the *Filtering behaviour* and *Managing measures* chapters.

Degenerate dimensions occur when an attribute used for filtering or aggregation is retained within a fact table, rather than being elevated to a standalone dimension. This typically occurs for one of two reasons:

- The attribute lacks sufficient business weight to justify its own dimension.
- The attribute has a high cardinality, making it computationally expensive to be a dimension.

Despite these pressures, degenerate dimensions are best avoided because they violate the natural filtering paradigm of dimensional modelling in Power BI, where dimensions filter facts and not the reverse. Retaining such attributes in fact tables can complicate user interaction and introduce undesired filtering behaviour. It also signals a failure to properly articulate business meaning when the column actually deserves to be its own dimension but is not.

If the attribute must be retained for display or cross-filtering, a transaction dimension may be preferable.

Conclusion

One common theme of this chapter is to push information back from lower-grain than the entity, back to the entity level, and then further abstracted to the dimension. The combination, choice and Sankey dimensions all follow this pattern. They take detail-

level complexity and express it as a dimension with a manageable set of values that users can interact with easily.

[DIAGRAM... Finer grain than entity → Entity → Reference table]

This movement is powerful. It makes the entity of interest the common denominator for all joins without shifting the grain. It avoids the chaos of inconsistent grains and the risk of double counting. The same principle applies to the storytelling dimension, which condenses any amount of detail into a single narrative at the entity level.

The purpose of this chapter is not to define an exhaustive list of all possible artefacts in a dimensional model. Ultimately, each table is either a dimension or a fact, and the sub-categorisation does not significantly change this.

Rather, the purpose of listing out these elements is to help data engineers shift to a mindset of seeing dimensions and facts through usage scenarios, and to recognise that they can play functionally different roles depending on the interactivity requirements.

In the end, what counts is that a user can place a value on the canvas and get a desired effect intuitively and reliably. This, rather than any technical classification of facts and dimensions, is the heart of dimensional modelling in Power BI.

Filtering behaviour

Filtering is the driver for an interactive user experience in Power BI. Filtering occurs when a slice of the model's data is selected for display or calculation based on a user action.

Filtering can also be complex. If used inappropriately, the model can quickly become frustrating or error prone. Consequently, data engineers need to have a deep, almost second-nature understanding of how Power BI filtering works in practice, and the options available for different scenarios.

Newcomers to Power BI often use bidirectional relationships incorrectly. An early and deep understanding of Power BI filtering behaviour will allow data engineers to use the right configuration instead of bidirectional relationships.

Ways of filtering

There are five common ways of filtering in Power BI:

- Directly through a table relationship
- Indirectly through common filtered tables
- In a table by a non-blank measure value
- In a visual by a visual-level filter
- On-demand through a measure formula

Directly through a table relationship

This is the standard case of filtering. When a table *X* filters another table *Y*, a user who selects a value from *X* will narrow down to the rows in *Y* based on the primary key and foreign key relationship.

The most important case of table relationship is when dimension tables filter fact tables. Dimension tables are typically small in cardinality, and the resulting relationship is highest performing.

New engineers often ask, "Why do we need dimensions at all? Why not leave all the information in the fact table? It is simpler." That is, why not rely solely on degenerate dimensions? There are three reasons for this.

First, nonexistence is impossible to express with degenerate dimensions. For instance, if the full set of reference values are "Red", "Blue", and "Yellow", but the transaction table does not have "Yellow", then this absence cannot be communicated without a dimension that has the full list.

Second, a proper dimension table is important for expressiveness. If the attributes are embedded in a fact table, the set of attributes, such as demographic details, become lost in the fact. Indeed, even the grain becomes unclear when attributes are

denormalised into a fact table. A user would have to read every single row to reverse engineer the entity and its grain.

Third, cross filtering or drill through of multiple processes are impossible with degenerate dimensions. The simplest example is that of the calendar. Without a calendar dimension, each fact table would have its own business date for its business process, and these cannot be used to compare the measures between two processes. Similar applies to other attributes such as region or product.

Indirectly through common denominators

Filtering also propagates indirectly — through shared dimensions across multiple tables. This behaviour is often missed by new engineers but is just as essential.

Consider two dimension tables *Dim1* and *Dim2*, and a fact table *Fact1*. Suppose *Dim1* filters *Fact1*. Now consider two cases:

- If *Dim2* does not filter *Fact1*, dragging *Dim1* [X] and *Dim2* [Y] into the same visual causes an error: “Can’t determine a relationship”. In this case, Power BI does not produce a cross product of all results.
- If *Dim2* does filter *Fact1*, Power BI returns only combinations of [X] and [Y] that share a row in *Fact1*. This behaviour is visible in the DAX query via the Performance Analyzer.

If multiple fact tables (*Fact2*, *Fact3, Fact4*, etc) are filtered by both *Dim1* and *Dim2*, then a visual containing *Dim1* [X] and *Dim2* [Y] will show combinations where any fact table has a matching row.

[SCREENSHOT PERFORMANCE ANALYZER]

This “upward” filtering from facts to dimensions enables the intuitive experience of a dimensional model. It is why dimensions alone can serve as the interface, even when facts are hidden. Mastering this behaviour is central to dimensional modelling.

In a table by a non-blank measure value

The previous method relies on implicit filtering via relationships. In simple models, this suffices. In complex models with five or more fact tables, it may not.

Measures can be used to explicitly control filtering.

Suppose *Dim1* filters *Fact1*, and a measure [Measure1] counts rows in *Fact1*. Dragging *Dim1* [X] into a table visual shows all values of [X]. But when [Measure1] is added, only values of [X] with a nonblank [Measure1] remain.

If [Measure2], [Measure3], etc. are added, the visual retains only values of [X] where any measure returns a value.

This mirrors fact table filtering, but with greater control. The data engineer defines which values are retained, rather than relying on implicit row existence.

[SCREENSHOT PERFORMANCE ANALYZER]

In a visual by a visual-level filter

Sometimes the measure cannot be placed directly in the visual, as with slicers, or doing so would interfere with layout.

In these cases, the report developer can apply a visual level filter using the measure value. This allows filtering without altering the visual's structure.

[SCREENSHOT]

If possible, this method should be avoided. The evaluation cost can be significant on a visual with a large number of rows for which to calculate the measure value.

On-demand through a measure formula

The previous methods allow the report developer to control which dimensional values are displayed in a visual. The prototypical context is the columns in a table visual, but also the columns and rows in a matrix visual.

[SCREENSHOT]

There are occasions when the data engineer needs to apply filters directly on the data that feeds a measure. A measure, unlike dimension values, is not a fixed value in the data. It is defined by a formula and returns only one value at a time. The prime example is a visual card.

[SCREENSHOT]

Usually, the need to apply a specific measure level filter arises because the standard model relationships do not support the intended outcome. In this circumstance, the filtering context must be created inside the measure definition.

Power BI supports this through DAX expressions such as USERELATIONSHIP and CROSSFILTER. These expressions allow the data engineer to activate a relationship between tables on demand, even if that relationship is not defined in the model schema.

Filtering scenarios

The five filtering patterns above can be combined to support a wide range of interactivity goals, all within the standard dimensional model framework of dimensions filtering facts using single directional relationships.

Displaying unit records (single business process)

A common requirement is for business users to view all the details of a business process. Suppose we have a business process represented by the *'Sales'* fact table. In

the ideal setup, the *'Sales'* fact table is hidden and represented through its dimensions and a single measure.

The set up is as follows:

- A fact table '*Sales'*
- The ID dimension *'Sales ID'* with business key [Sales order number]
- The calendar dimension *'Sales calendar'*
- The business dimension *'Sales product'*
- The measure [Total sales amount] which sums up the sales amount in *Sales*

When you place dimension fields together with *'Sales ID'*[Sales order number], Power BI returns only combinations of dimension values that correspond to existing rows in the fact. Since *'Sales ID'*[Sales order number] is the business key for a sales transaction, it uniquely identifies the row in *'Sales'*. Adding *'Sales ID'*[Sales order number], *'Sales calendar'*[Sales date], and *'Sales product'*[Product name] returns the complete transaction context for each sale.

Sales also have an amount recorded in *'Sales'*[Sales amount] on the fact table. A simplistic approach is to place this column on the canvas, treating it like a degenerate dimension. A better approach is to use the measure [Total sales amount]. Because *'Sales ID'*[Sales order number] uniquely identifies a single row, [Total sales amount] returns the correct result as the sum of that one row in the current context, which is the same numeric value as *'Sales'*[Sales amount] for that transaction.

While this sounds complex in the backend, it is seamless for the user. In the ideal setup, the fact table and thus *'Sales'*[Sales amount] is hidden. The user only sees the measure [Total sales amount] and does not see both the measure and the raw *'Sales'*[Sales amount] column, which removes ambiguity.

This is far superior to exposing both the measure and the raw column. In this instance, both would show the same number for each transaction, which encourages users to treat them as interchangeable. In other contexts such as charts, aggregations, and time intelligence, only the measure behaves correctly. That inconsistency is confusing when something appears to work in one visual but not in another. With this setup, [Total sales amount] behaves intuitively across all situations, so users get consistent results without needing to remember special cases.

It is important to consider how the user experience changes depending on the order in which fields are selected:

- **First *'Sales ID'*[Sales order number], then *'Sales calendar'*[Sales date]**
  This immediately gives the valid combination of sales order numbers and sales date because the combination of dimension values is filtered by their common

facts (second method of filtering). This gives desired unit-level transaction record because the business key selects the correct grain.

- **First *'Sales ID'*[Sales order number], then [Total sales amount]**
  This also immediately gives the unit-level transaction record. However, this time it is not because of the common fact (second method of filtering), but because [Total sales amount] returns a non-blank measure value for the [Sales order number] (third method of filtering). If there are sales order numbers without a sales amount (for example, unconfirmed sales), those order numbers will not appear. This is, for most use cases, the desired outcome.
- **First [Total sales amount], then *'Sales product'*[Product name]**
  This initially returns the total sales amount for the entire model as a single number, then breaks it down by product. When the user adds *'Sales ID'*[Sales order number], the table expands to the unit record. In other words, the user experience is "correct" at every step.

What we see is that, regardless of the sequence of field selection, the model produces sensible business results at every step. This is the power of a properly designed dimensional model focused on dimensions and measures, with single-direction filters on facts.

This seamless experience would not occur if the user had selected the degenerate *'Sales'*[Sales amount] instead of [Total sales amount] in the last scenario. The user would first see a distinct list of sales amount values, then an incomprehensible combination of sales amount and product, and only after adding *'Sales ID'*[Sales order number] would the table make sense. The experience would be jarring rather than seamless. This is another reason to hide *'Sales'* altogether and avoid exposing any degenerate numeric values.

There is also an additional consideration of what happens when there are existing filters on the canvas, such as a filter on product name or date. While all the previous sequences still work, the most seamless outcome occurs when the user starts with the measure [Total sales amount], as in the last sequence. We do not expand on this here and leave it as an exercise for the reader.

Even though this is a simple use case of presenting transaction records for a single business process, there is careful modelling work behind the scenes to create a seamless experience that caters to a staggering number of possibilities. All these possibilities are elegantly handled if the data engineer uses the best practice of focusing on dimensions and hiding facts.

In this example, *'Sales ID'*[Sales order number] is a single column business key. If the business key were multi-columns, then all columns need to be used.

Displaying unit records (two business processes)

Continuing with the previous example, consider the situation where sales can be refunded, modelled as follows:

- A fact table *'Refund'*
- The calendar dimension *'Refund calendar'*
- Refunds can be tracked to the original sales order number and are filtered by all dimensions that filter *'Sales'*
- There is no business key for a refund, and hence no *'Refund ID'*
- A measure [Total refund amount] which sums the refund amount. Because there can be partial refunds, this does not equal [Total sales amount] even for the same sales order

[Screenshot]

In this model, the user may want to see the transactions for sales, for refund, and sometimes for both simultaneously.

Displaying sales

When displaying records for sales, everything works as if there were only the *'Sales'* fact table. This is because Power BI shows combinations of dimension values if there are valid rows in any of the fact tables they share. Since every refund comes from a sale, the *'Sales'* fact table is a superset of *'Refund'*, so adding the refund fact table does not cause an issue.

The fact that *'Sales'* is a superset of *'Refund'* is essential. If there were [Sales order number] values in *'Refund'* that do not exist in *'Sales'*, then selecting *'Reporting calendar'*[Reporting date] and *'Sales ID'*[Sales order number] would not correctly identify the list of sales transactions.

Furthermore, because filters are single-direction, even if the report canvas has a filter on *'Refund calendar'*, this filter does not interfere with the presentation of sales transactions. This would not be the case if there were bidirectional filters, which would return unexpected results.

Displaying refunds

When displaying records for refunds, the user is interested in [Total refund amount], but it is also necessary to display the sales context of *'Sales ID'*[Sales order number] and *'Sales product'*[Product name]. If the user selects the [Total refund amount] column first, this would return the total measure value for the entire dataset and then adding the sales attribute dimensions would break this number down until the ID dimension is selected to return the unit-record transaction.

If the user selects sales attributes first, Power BI displays the full list of sales orders and their products, regardless of whether they were refunded. When the user adds either

'Refund calendar'[Refund date] or the [Total refund amount] measure, the table filters down to refund transactions. This happens because these attributes are linked only to the *'Refund'* table, not *'Sales'*.

Whether the user selects the sales attribute first, and then the unique refund attributes or in the reverse order, a dimensional model gives a seamless and intuitive experience for the user.

Nevertheless, the situation is not so simple as it appears. Consider a case where a sales order of $100 is refunded in two partial refunds of $50 on the same day. Because there is no primary key for refunds, the refund for that sales order appears as a single transaction with [Total refund amount] = $100, rather than two rows of $50. For most scenarios, this may be sufficient, but in some cases, it is not.

When it is necessary to see each transaction row as in the business process, the solution is to create a key that distinguishes each refund transaction. One option is to create a surrogate key [Refund SK], a unique integer for each refund, and expose this as *'Refund ID'*[Refund SK]. However, this is not preferable for two reasons:

- [Refund SK] is high cardinality, increasing model size due to the *'Refund ID'* dimension and relationship columns.
- [Refund SK] has no natural business meaning and would require additional learning curve.

A better approach is to create a [Refund number], which is a sequence number for each refund action for a sales order, to form a composite primary key of [Sales order number] and [Refund number]. This effectively identifies each refund. The [Refund number] should be exposed *'Refund ID'*[Refund number], which is a distinct list of integers with the maximum number equal to the highest number of refunds in a sale. This is a partial ID dimension because it has one part of the full key to identify a refund. This dimension is low cardinality, has minimal resource impact, and is natural for users. From the user's perspective:

- Using [Total refund amount] gives the refund value for the transaction.
- Adding *'Refund ID'*[Refund number] breaks this down into separate refund transactions.

This is perfectly natural for the user where every step returns a meaningful result.

The use of the measure [Total refund amount], rather than the degenerate '*Refund'*[Refund amount], is essential for correct result. Suppose the user selects the raw value *'Refund'*[Refund amount] instead. In the absence of a refund primary key, Power BI would display the previous case as one single row of $50 for the sales order—

an incorrect result. This is yet another example of why exposing degenerate numeric columns is problematic and why the fact tables are better hidden.

Displaying both sales and refund

The user may want to see details of the sales and refund business processes simultaneously. There are three possibilities:

- The processes displayed side by side
- The sales transaction with supplementary details about refund
- The refund transaction with the original sales amount

Displaying the refund and sales processes side by side on the same report page is straightforward.

Suppose there is a table visual for sales with the columns:

- *'Sales ID'*[Sales order number]
- *'Sales product'*[Product name]
- *'Sales calendar'*[Sales date]
- The measure [Total sales amount]

And a table visual for refund with the columns:

- *'Sales ID'*[Sales order number]
- *'Sales product'*[Product name]
- *'Refund calendar'*[Refund date]
- The measure [Total refund amount]

These two tables work well because of the dimensional model design.

If there is a slicer on *'Sales product'*[Product name], and the user selects a particular product name, then both visuals filter down to that product.

Similarly, if there is a slicer on *'Sales ID'*[Sales order number], the user can look up a specific sales order number and retrieve details of both the sale and any refund. In addition, a user can click on a row in the sales table and cross-filter to any refund. This works because *'Sales ID'* is a conformed dimension for both Sales and Refund facts.

For the business user, this is exceedingly convenient navigating unit-records. This is why the ID dimension is one of the most powerful dimensions and an indicator of a good model. The same cannot be achieved if [Sales order number] were used as a degenerate dimension. If degenerate dimensions were used, the experience would be highly frustrating for the user because none of the filtering would work as expected.

The user experience is perfect when operating on the conformed dimensions *'Sales ID'* and *'Sales product'*. However, it becomes complicated with the two calendar dimensions: *'Sales calendar'* and *'Refund calendar'*.

Users expect to slice by time. Neither calendar works perfectly:

- A slicer on *'Refund calendar'* has no effect on the sales table because it does not filter the Sales fact.
- A slicer on *'Sales calendar'* correctly filters the sales table, but for the refund table, it returns refunds for products sold on that date—not refunds that occurred on that date.

In simple examples, users may interpret this correctly. In real-world scenarios with multiple dates, this nuance is often difficult to understand. In the worst case, it would silently mislead users.

The solution is to create a role-playing *'Reporting calendar'* that links [Reporting date] to both *'Sales'*[Sales date] and *'Refund'*[Refund date]. When used as a slicer, both tables return rows for that date. This works because the selection filters the Sales and Refund facts directly (first method of filtering), and then the visuals filter out combinations without matching fact rows (second method of filtering). This chain of filtering behind the scenes is responsible for creating a seamless experience for the user.

If, instead of having two tables side-by-side, the user wants a single table of sales transactions with refund amount, it suffices to use the sales table columns above and add [Total refund amount] as an additional column. The table visual then has two measures [Total sales amount] and [Total refund amount]. The result is that:

- All sales transactions display as normal, with the [Total sales amount]
- Rows with refunds show a value in [Total refund amount]; others remain blank.

The outcome works because Power BI displays rows where *any* measure is non-blank (third method of filtering). In addition, if a refund exists without a sales amount, the row stands out with a blank [Total sales amount] and a non-blank [Total refund amount].

However, the user cannot add *'Refund calendar'*[Refund date] to this visual. Technically, this dimension relates only to *Refund*, not *Sales*. The business interpretation for this is because one sale can have multiple refunds on different days, and therefore it is not logically correct to add the refund date when the intent is to display one row per sales. If displaying refund dates is necessary, the data engineer can create a measure [Sales refund dates] that returns the unique refund date if there is one, or concatenates distinct dates if there are multiple.

The scenario with displaying sales does not mirror to refunds. Adding [Total sales amount] to the refund table causes Power BI to return incorrect results by cross-joining sales and refunds. This happens because *'Refund calendar'*[Refund date] relates only to Refund and does not filter the Sales fact. There is no valid relationship path for [Total sales amount] to work.

Thus, users expect to drag [Total sales amount] into the refund table and see the original sales amount but instead get cross-join of rows. This is one of the situations where the default dimensional model is not intuitive out of the box and requires additional tuning.

There are multiple solutions while staying within the parameters of dimensional modelling:

- Define [Total sales amount] to behave differently depending on context using DAX functions like TREATAS.
- Denormalise *'Sales'*[Sales amount] into the Refund table as [Sales amount before refund]. This remains intuitive because a keyword search for "sales amount" shows both measures. Clear naming and display folders grouped by business process help disambiguate.

Since the *Refund* fact may have multiple rows per sale, [Sales amount before refund] is not a simple sum. The DAX is more complex and is explained in the *Designing measures* chapter.

The first option of adjusting [Total sales amount] gives the most seamless experience for the user at a complexity and performance penalty for the measure itself. It is not advisable to complicate such a core business measure to satisfy a narrow use case.

The second option adds a more narrowly defined [Sales amount before refund] measure to the model. The measure is unlikely to be useful in other scenarios. However, it is more explicit, remains intuitive for the business, and maintains lower DAX complexity overall.

Displaying unit records (three or more business processes)

The techniques for displaying sales and refunds rely on conformed dimensions and the existence of fact rows to filter. In this, the ID dimension plays a special role.

This does not scale neatly. With three or more facts it becomes hard to know which combinations of dimension values still have at least one row in any fact. Managing multiple ID dimensions is also complicated because some are conformed against some facts but not others. Many processes will not have a ready business key.

A practical solution is to control the display with measures. The data engineer can create one display measure per business process that returns a value for that fact:

- [Display sales transaction]
- [Display refund transaction]
- [Display … transaction]

Each measure returns 1 when there is at least one row in its fact under the current filter context. These should be placed collectively in a display folder named “Display unit records”. A refinement is to return a value only when *'Sales ID'*[Sales order number] is filtered so that unit records appear only after an order is selected. This can be done by checking the calculation for *'Sales ID'*[Sales order number] context using ISINSCOPE.

[SCREENSHOT]

Adding one of these measures to a table visual filters dimension values to those that have a matching fact row (third method of filtering). The user can hide the helper column by renaming it and shrinking its width to zero.

[SCREENSHOT]

An alternative is to use the measure as a visual level filter the fourth method of filtering (fourth method of filtering). This is convenient but has noticeable performance penalty on large facts.

[SCREENSHOT performance analyzer]

While this has a learning curve for the user that involves a Power BI trick, it is a helpful clarification. A model that has more than four or five business processes are likely to be complicated to understand. An explicit selection of business processes through the “menu” of display unit-records measures can aid mental clarity.

Cascading filters

Cascading filters refer to the behaviour where selecting a value in one filter narrows the choices in another so that only valid options remain. For example, if a report page has a slicer for *'Sales product'*[Product name] and another for *'Sales calendar'*[Sales date], then choosing a product limits the dates to those where that product was sold, and choosing a date limits the products to those sold on that date.

[SCREENSHOT]

The first method of filtering requires tables to have a chain of relationships that flow in from one to the other. In this case, no such path exists between the two dimensions. The second and third methods require two elements in a single visual—a dimension value and either another dimension or a measure—so they do not apply either.

The solution is to use a visual level filter, which is the fourth method of filtering. In this case, the measure [Total sales amount] can be used as a visual level filter. A more explicit option is to create a measure such as [Has sales] which returns 1 if the count of

rows in the *'Sales'* fact table is greater than zero.
Therefore, the solution is to rely on a common fact table and a measure that can pass information back to the dimension against the default filter direction.

[SCREENSHOT]

Aggregating dimension values
Cascading filters aim to filter one dimension by another. Sometimes the goal is to aggregate a dimension by another dimension.

Following the example of sales and refunds, suppose the user wants to display a concatenated list of distinct products refunded on each day. If [Product name] were already included in the *'Refund'* fact table as a raw value, the measure would take the distinct values in *'Refund'*[Product name], and apply CONCATENATEX on those values.

[SCREENSHOT]

However, this approach is often not possible. For example, if *'Sales product'*[Product name] is linked to *'Refund'* via [Product ID] rather than the product name, or if the user wants to aggregate another column from *'Sales product'* such as [Product type]. It is not practical to include all such columns in the fact table.

An inexperienced data engineer might denormalise the required column into the fact table or even precompute string concatenation in the data layer so that the column can be used as a degenerate dimension. Sometimes it is pragmatic to include a frequently aggregated column in the fact table. In general, this is not desirable because it is heavy work to modify a large fact table for every column the user might want to aggregate. Pre-aggregation is also not ideal because it cannot respond to user filter context.

Instead, the solution is to notice that the information is *already* in the model, and it is just a matter of retrieving it. The fifth method of filtering—on-demand filtering in a measure—provides the answer. The DAX is to concatenate *'Sales product'*[Product name] and wrap it in a CALCULATE context where CROSSFILTER is set to both directions. This temporarily changes the single direction relationship to bidirectional and allows *'Refund calendar'* to filter *'Sales product'*.

[TEST , SCREENSHOT]

Filter by having
Consider the case of an organisation's HR system. The dimensional model is set up as follows:

- A dimension *'Employee'* that stores details such as name, email, and date of birth. This table is type II to reflect changes in employee attributes. The key columns are [Employee ID] and [Start datetime]. An [End datetime] signals the

end of the validity period of a row. [Employee SK] is used as a surrogate key for Power BI relationships.

- A dimension *'Organisation unit'* that stores the hierarchy of teams in columns [Department name], [Group name], [Branch name], and [Section name]. This table is also type II because names and hierarchy can change. The key columns are [Team unit ID] and [Start datetime]. An [End datetime] signals the end of the validity period of a row. [Team unit SK] is used as a surrogate key for Power BI relationships.
- An end-of-month fact table *'Employee end of month'* that stores the employee's attributes and organisation unit at the end of each month.

[SCREENSHOT]

The user wants to search an employee's and organisation unit history by personal attributes such as name or role. Since *'Employee'* is type II, a direct search returns only the end-of-period states that match the point-in-time attribute values. This is not the desired outcome. The user wants the full history of the employee using any historical attribute.

This is an example of filtering dimensions by whether they have an attribute, rather than filtering on the attribute value itself. We call this **filter by having**. It applies to any scenario where a user would like to look up a header on a detail attribute and return all other details for that header. New data engineers often try to solve this by restructuring the fact table. This leads to complex code and artefacts that confuse other contexts.

A simpler solution is to add a search dimension to the model, *'Employee search'*. This can be a duplicate of *'Employee'* and relates to *'Employee'* on [Employee ID] in a snowflake configuration. The relationship is many-to-many and single direction.

[SCREENSHOT]

Using *'Employee search'*, the user can look up any attribute of the employee at any point in time, retrieve the [Employee ID], and return all rows in *'Employee'* that match this ID.

Under this view, *'Employee search'* and *'Employee'* work together as one logical "dimension" that acts as an interface for searching the model. This pattern generalises. The data engineer can add *'Compliance search'* or any other derived attribute of [Employee ID] as additional search dimensions as more branches of the snowflake. From the user's perspective, these are simply "search" tables that allow searching employees by their history. Unlike tampering with fact tables, this solution is non-destructive. It adds a few tables as an interface without modifying the rest of the model.

Dynamic type I

Type II dimensions show an entity at its point-in-time attribute. Type I shows the latest attribute value. **Dynamic type I** shows the latest attribute value as at a user-selected time. This is the natural expectation in entity tracking.

s 22(1)(a)(ii)

Consider employees and their organisation unit. The fact *'Employee end of month'* returns employee attributes and organisation unit at the end of each month.

Suppose the user wants employee metrics for a group over a period, such as the total number of commencements and separations in the last 12 months. This is not the count as at the month end. It is all commencements and separations that happened in the 12 months prior.

The *'Organisation unit'* dimension is type II, so the fact reflects the hierarchy as at each period. When units reshuffle often, teams that moved between groups do not bring their counts with them. A measure of commencements broken by [Group name] gives the wrong result. It only shows the hierarchy as at now and ignores teams that were in the group during the last 12 months.

If the requirement were fixed to the latest view only, a type I version of *'Organisation unit'* would suffice. However, the goal is to report against the hierarchy as it stood at the end of every 12 months period. This requires using the view of the organisation unit to change dynamically as selected by the user, and hence dynamic type I is needed.

s 22(1)(a)(ii)

The problem is notoriously complex to solve through amending the fact table. It requires associating all the organisation unit's history to each fact row.

Rewriting facts to carry the full hierarchy history is complex and fragile. The model already has the history in *'Organisation unit'*. The solution is to surface it at query time.

Two steps implement this approach

1. **Broaden the relationship for evaluation**
   Relate each fact row not to a single point-in-time row in *'Organisation unit'*, but to all rows for that [Team unit ID]. Use a many-to-many, single-direction relationship on [Team unit ID]. Each fact row now has the full history of its team available during calculation.
2. **Select the valid row at query time**
   In the measure, take the latest date in the current context and filter *'Organisation unit'* to the row whose validity covers that date: [Start datetime] ≤ selected_date < [End datetime]. For a given [Team unit ID] this is unique because the primary key is [Team unit ID] and [Start datetime]. Under this filter the table behaves like a one-to-many join for that evaluation and the measure returns values only for

the valid row. Other rows go blank and drop out by the third method of filtering on non-blank measure values.

This alters how *'Organisation unit'* behaves. There are two options that preserves the behaviour of the dimension.

Option A. Add a new dimension

- Duplicate *'Organisation unit'* as *'End of period organisation unit'*.
- Add [Team unit ID] to *'Employee end of month'*. The fact now carries [Team unit SK] for the normal join to *'Organisation unit'*, and [Team unit ID] for the many-to-many join to *'End of period organisation unit'*.

  [SCREENSHOT of model relationship]

- Wrap measures when dynamic type I is needed. For example, rename [Employee commencements] to [_Employee commencements]. Define [Employee commencements] that evaluates [_Employee commencements] inside CALCULATE, filtering *'End of period organisation unit'* to the valid row as at the latest date in context.

```
Leave taken =
var selected_date = max('Calendar'[Date])
return calculate([_Leave taken]
                , keepfilters(
                                and('End-of-period org'[Start date] <= selected_date
                                    , selected_date< 'End-of-period org'[End date] )
                                )
                )
```

[SCREENSHOT of DAX]

In this option, the user chooses perspective by choosing the dimension.*'Organisation unit'* gives type II. *'End of period organisation unit'* gives dynamic type I.

Option B. Keep one dimension and switch in measures

- Keep the normal one to many single direction relationship on [Team unit SK].
- Add a many to many single direction relationship on [Team unit ID]. It remains dormant.
- In measures that need dynamic type I, deactivate the SK path and let the [Team unit ID] path apply for the calculation. Then filter to the valid row as above.

  [SCREENSHOT of DAX]

In this implementation the dimension works as type II by default, and the user can choose the dynamic type I perspective on certain measures.

Under both options the original dimension *'Organisation unit'* keeps its normal behaviour. All existing measures work as before. Both options are non-destructive. The distinction between them is whether the user chooses type II or dynamic type I through using different dimensions, or through using different measures.

Buttons and effects

These scenarios are not intended to be an exhaustive list of filtering techniques or scenarios – this would be a book itself. Instead, their purpose is to illustrate a philosophy of seeing Power BI as an interface that users click to achieve an effect. In this philosophy every table has one of two roles:

- Tables that define the interface as buttons to click, usually the dimension tables
- Tables that define the content which drives the effect, usually the fact tables

A good model keeps these roles separate. Every table serves a single purpose and not a mix. This is why an indicator of quality is that all fact tables are hidden.

The first three examples show how dimensions present information from facts using the more difficult case of displaying transaction records. They illustrate that the fact tables, which are the natural focus for new data engineers, are *not* the artefacts to expose to the users. In fact, trying to expose the fact tables and degenerate dimensions will only cause issues. Instead, the dimensions act as the entry point to the data.

The same philosophy explains why degenerate dimensions and bidirectional relationships frustrate users. They blur the line between tables that act as buttons and tables that create the effect. The examples of cascading filters and aggregating dimension values show that the data engineer can handle occasions where a dimension is on the receiving end of the button and effect relationship without compromising the model with degenerate dimensions or creating bidirectional relationships.

The final two examples generalise the idea of a dimension to a set of tables. For this purpose, a dimension is not simply a lookup table. If the user sees it as intuitive, and get the desired outcome unambiguously and rapidly, that is what matters. In this view, a dimension's chief purpose is to be the user's interface to the model. The final two examples also demonstrate Power BI's flexibility of using many-to-many relationships that would otherwise require substantial modification to the model.

This is the philosophy of separation of concerns from software design. It maintains a clear distinction between the interface layer and the content layer. Power BI supports this through the five filtering mechanisms described at the start of the chapter. This

perspective pushes the data engineer to see a data model as software and to design a clean interface that is business-centric and intuitive for the user.

Designing measures

Measures are the face of facts. If dimensions are the interface of the model, measures present the effect of that interaction. Fact tables provide the content. Measures turn that content into answers. As explained in *Filtering behaviour*, measures also give the user an explicit way to control filtering. They are part of the interface and deserve deliberate design. For this purpose, Power BI's implicit calculations can do more harm than good. For this reason, **Discourage implicit measures** should be turned on.

This chapter covers three areas. First, indicators of good measures. Second, the four types of measures. Third, basic technical patterns for building measures. The next chapter, *Measure of measures*, introduces techniques for managing many measures in a structured way.

Indicators of good measures

There are two indicators of good measures, business centricity and technical simplicity.

Business centricity

Every aspect of a measure should reflect business meaning. This includes its definition, name, its hover text, its display folder, and its placement alongside related measures.

Alignment to business reality

When designing measures, one of the most important criteria is alignment to real-world events. A common mistake is to perform a distinct count on a database key and assume it reflects a meaningful metric — even when the key has no real-world correspondence.

For example, a data engineer might define a measure like [Inspection count] as the distinct count of [Inspection ID] in the *Inspection* fact table. However, [Inspection ID] may simply be a system-generated key used for record retrieval. It carries no inherent meaning. In the future, the system design might change its implementation to split one inspection into ten rows or merge ten into one for record-keeping purposes. This does not reflect a genuine change in inspection effort. The measure is misaligned with reality.

Instead, every measure should quantify real-world business events that users recognise and care about. In this example, the data engineer could:

1. Measure the time taken to conduct an inspection
2. Measure the number of entities inspected

The second option is particularly useful across a wide range of scenarios — whether counting entities inspected, entities failing a criterion, or entities reaching a milestone. It is easily implemented when the entity of interest has been clearly defined from the

outset, and when the pipeline consistently computes and aggregates information at that grain. This is one of the purposes of the *Reduce* step in the pipeline.

Business explicit

The definition of a measure should be explicit on its business meaning, rather than generic. A common misstep among new data engineers is to create a single technical measure—such as [Employee count], a distinct count of employees in an end-of-period fact table—and assume that users will derive all other insights through manual filtering. This approach places undue burden on users, requiring them to memorise arcane filter combinations to arrive at answers. A more effective model presents a suite of measures that surface answers directly.

In the case of employee metrics, this suite might include:

- [Active employees end of period] — employees with an "active" [Employment status] at the end of each period. If unambiguous, this may be shortened to [Employees end of period].
- [Separated employees end of period] — employees with a "separated" status at the end of the period.
- [Current number of employees] — the latest value of [Active employees end of period], suitable for grab-and-go use.
- [Current employees on long service leave] — a subset of [Current number of employees] filtered by [Is on long service leave].

These examples illustrate the breadth of possible measures beyond a generic [Employee count]. Requiring users to manually filter for each scenario is both error-prone and frustrating.

In most cases, a technical measure such as [Employee count] which is a distinct count of all employees across the organisation's history, offers little business value. If not directly useful, such measures are better omitted. Where technical measures are needed to support other calculations, they can be hidden from the interface and prefixed with an underscore. For example, [*_Employee count]* and [*_Employee count] end of period* may serve as internal components for more meaningful outputs.

Names

Names must be explicit, grammatical, and able to stand alone. Measures can be dragged anywhere on the canvas, so the name should carry its context. Prefer [Employee count] to [Count]. Avoid non-standard abbreviations. Saving a few characters does not help the user or the next engineer.

Description

Hover-text descriptions should also reflect business meaning. Users should not be expected to accept complex logic without explanation. As the first point of contact for many users, the measure description is an opportunity to build trust. Descriptions should articulate the business logic behind the measure, not restate the DAX syntax. For more intricate cases, near-pseudocode in business language may be appropriate.

Display folder

Display folders offer a practical way to reinforce business centricity. Measures can be grouped by business process, while technical or dashboard-specific measures such as those for colour formatting or data currency can be placed in folders like “Dashboard” or “Data currency.”

Proximity to other measures

The business centricity of a measure is influenced by its proximity to other measures. The two ways of proximity are alphabetical placement within a display folder and keyword-based lookup in the field list. These interactions warrant deliberate consideration from the data engineer.

When similar measures are defined explicitly, they help users form a clearer mental model of the data. For example, the presence of [Separated employees end of period] reinforces the meaning of [Active employees end of period], reminding users of the separated employees that may have been unconsciously forgotten. Likewise, [Current number of employees] clarifies that [Active employees end of period] is a point-in-time measure. These compare-and-contrast relationships benefit from thoughtful naming that places related measures adjacent to one another.

A data engineer can take advantage of *the proximity effect* by consciously placing these compare-and-contrast measures next to each other through alphabetical naming.

Technical simplicity

Measures in Power BI should be technically simple. The foundations for technical simplicity are laid when building the data pipeline, not when designing the dimensional model. There are two ways to achieve this – precomputing complex information and relying on binary flags for simplification.

Precomputing complex information

Complex information should be computed and stored in the data layer. This frees the measures from the burden of handling business logic and allows them to focus solely on supporting user interaction. If the DAX is non-trivial, it is a sign that the model design may be sub-optimal.

Throughout *Creating information*, the emphasis is on creating meaningful fragments that can be used as plug-and-play by users. For instance, if [Is on long service leave] is a

complex calculation that requires windowing across an employee's attendance, then this calculation can be precomputed in a fragment called *HR.EmployeeLeave*.

The most important case of precomputing complex information is that of summary dimensions. In many scenarios, the complexity of information comes from having a lot of finer details in subprocesses for an entity of interest. Precomputing these, and pushing their expression to reference data tables, allows the data engineer to access all this complex information without repeating it in DAX. Of the summary dimensions, the most important is the storytelling dimension. These are explained in the *Storytelling* chapter.

Precomputing complex information can be overdone. Data engineers who are unfamiliar with the features of Power BI can end up repeating heavy computation that could be extracted from the model data itself with the appropriate configuration or DAX definitions. More significantly, precomputing information can lead to loss of interactivity, such as pre-aggregated string values that can no longer be filtered down. In some cases, precomputation can even lead to wrong results, such as taking averages of averages rather than using the unit-record data in the model to calculate non-additive measures.

In general, precomputing complex information should not result in an awkward distortion of the dimensional model. Data engineers can avoid this by being familiar with the content of *Filtering behaviour* and the techniques introduced later in this chapter.

Preparing binary flags

Binary flags are columns with true or false values. Their main place is in the reference tables, and hence dimensions in the dimensional model. For example, an '*Employee leave'* dimension may have an [Is on long service leave] column to indicate whether the employee is on service leave at the end of the period of the '*Employee end of month'* fact table.

Binary flags have a special place in simplifying DAX formulas. As an example, [Is on long service leave] can replace text filters such as [Leave status type] in ("Long service leave", "Extended leave"). But they can also replace more complex calculations, such as checking the employee's attendance, by pushing this to the data layer.

Binary flags are preferred because, firstly, they are syntactically checked. A developer misspelling "Long service leave" in text does not result in a syntax error. However, the same spelling mistake in the column name [Is on long service leave] will result in an error. This is a small enhancement, but over a model with hundreds of measures, the possibility of spelling mistakes drastically adds up. Binary flags help mitigate the risk.

Secondly, binary flags are simple to use in a DAX formula. As a binary column, it does not require any equality operators. For example, instead of

```
[Current employees on long service leave] =
    CALCULATE([Current number of employees]
        , KEEPFILTERS('Employee leave'[Is on long service leave] = TRUE)
    )
```

The formula is

is simply:

```
[Current employees on long service leave]
    = CALCULATE([Current number of employees]
        , KEEPFILTERS('Employee leave'[Is on long service leave])
    )
```

The reverse can be expressed with a NOT, as in:

```
    KEEPFILTERS(NOT 'Employee leave'[Is on long service leave])
```

As a binary column, they participate elegantly in AND, and OR operators.

[SCREENSHOT]

As a binary column, they participate elegantly in AND and OR operators.

Using binary flags, the measures are faster performing, easier to maintain, and easier to understand. The business logic has been suitably abstracted so that each measure is self-explanatory.

Four types of measures

It is useful to distinguish four types of measures. These are not technical categories, but functional ones.

- Aggregating measures
- Dimensional measures
- Filtering measures
- Dashboard measures

Aggregating measures

Aggregating measures are the ones most people think of when they think of measures. These include counts of rows or sums of values against the fact tables. In more complex cases, they may need to respond to the user's evaluation context, such as calculating the number of employees at the end of a period.

Dimensional measures

Dimensions contain attributes that are generally used for grouping other measures. Dimensional measures are those where dimensional values become measure values themselves. One example was described in the chapter *Filtering behaviour* under the scenario *Aggregating dimension values*. In that example, the requirement was to display all products that had been refunded on a particular day.

It is important to recognise dimensional measures as a separate category for two reasons. First, the values are often drawn from the dimension tables rather than the fact tables. This means they are not subject to the usual filtering setup in a dimensional model and may require special adjustment in DAX formulas to support interactive behaviour.

Second, dimensional measures are often mentally confused with the dimension value itself. A dimension value can be used to cross-filter, and can be placed as rows or columns in a matrix visual. A dimensional measure, however, cannot be used for either. Its chief purpose is display and does not offer cross-filtering interactivity. That interactivity must come from other dimensions in the visual.

In the case of a degenerate dimension in a fact table, such as displaying a concatenated list of free-text comments in an annotation fact, it becomes academic whether the measure is dimensional or aggregating. The distinction is less important than recognising that attributes can become measure values as well, with its different filtering properties.

Filtering measures

As explained in *Filtering behaviour*, measures offer a way for the data engineer to explicitly control filtering. This is done through non-blank measure values and through use in a visual-level filter. Consequently, the model may contain measures whose sole purpose is to control filtering. These typically return either 1 or blank.

Examples include the [Display unit records] measures described in *Filtering behaviour*. Other examples may be [Has sales] or [Has employee leave], which also return either 1 or blank. The difference is that [Display unit records] are intended to be used strictly with the ID dimensions to list out business transactions. This enforcement may be done via the function ISINSCOPE.

Even in the case of filtering measures, they must remain meaningful for the user. The description should explain how the measure is to be used.

Dashboard measures

Dashboard measures refer to measures designed specifically for creating a visually attractive and informative report. These may include coloured arrows, icons, or expressions that display the user's selection context via SELECTEDVALUE. Power BI supports their use in a wide range of scenarios, including report titles, chart titles, and chart colours.

Technical patterns

The following introduces technical patterns that cover a wide range of usage scenarios. The first three fall under the standard case:

- Base measures
- Derived measures
- Dashboard measures

Then three more scenarios that are more complex:

1. Polymorphism
2. Hidden grain
3. Unsupported relationships

The next chapter will introduce an additional technical pattern using switch measures.

Measures in Power BI require unit testing. This is especially the case for more complex ones. This is covered in the chapter on *Automation*.

Base measures

Base measures interact directly with the fact tables in a straightforward way.

For measurable facts, these include SUMX, DISTINCTCOUNTNONBLANK, COUNTROWS, and similar aggregations.

For end-of-period facts, these include an additional detection of evaluation context to find the latest date in context and then evaluate an aggregate measure for that period.

For annotation facts, these include CONCATENATEX to display detailed textual values.

As explained in *Technical simplicity*, these measures should be simple and largely a direct aggregation of the fact tables, supported by a KEEPFILTERS on a binary column to narrow to specific business usage. For example, [Current employees on long service leave] should be derived from [Current number of employees] on the [Is on long service leave] column.

Derived measures

Derived measures are the next level from basic measures. These include ratios, time-intelligence, and population comparisons.

Ratios apply DIVIDE to two basic measures.

Time-intelligence applies standard techniques for calculating rolling periods, same period last year, and similar scenarios.

Population measures “escape” from the filter context to compare the measure with a larger population, such as comparing a team’s metric with the national metric. These can be achieved using functions such as REMOVEFILTERS.

Derived measures often follow repetitive patterns such as calculating a number for the same period last year. This could be better managed using calculation groups or switch measures.

Context-aware measures

Dashboard measures are one of the four types described earlier. They are designed to create visually attractive and informative dashboards. As such, they often need to “reach out” to inspect the filtering context that the user has chosen. For example, a report title may display the product name if one product is selected, or show “Multiple products” if more than one is selected.

DAX supports a wide range of context-awareness functions including ISFILTERED, ISINSCOPE, HASONEVALUE, and SELECTEDVALUE. They explicitly target what the user has selected. The functions VALUES and ISBLANK can be used to query dimensions to inspect what values exist in the filter context, which can inform the measure outcome. Another common pattern is to select the maximum reporting date from the model’s chief reporting calendar to reflect the latest date selected by the user.

For simple business processes, these patterns cover the bulk of usage scenarios. However, there are occasions when the dimensional model does not allow such straightforward implementation. The following scenarios allow a DAX solution without making drastic changes to the model.

Polymorphism

Polymorphism refers to the idea that the value of one column depends on the value of another. A common case is the header–details situation where actions can take place on either. For example, a business process may inspect travellers (header) and their bags (details) for contraband goods. The 'Inspection' table may record inspections on both. A column such as [Inspection item type] with values “Traveller” and “Bag” determines the meaning of [Inspected item SK].

In this scenario, a simple DISTINCTCOUNT of [Inspection item SK] will not give the total number of distinct items inspected. In a well-designed dimensional model, the determining column would be expressed in a dimension. For example, an 'Inspection type' dimension would contain [Inspection item type], while the aggregated value [Inspection item SK] is in the 'Inspection' fact table.

The solution is to create a temporary table using SUMMARIZE to group on the determining column, apply per-segment aggregation on the fact table, and then aggregate over the segments.

```
var inspection_per_type =
  SUMMARIZE(
      'Inspection type'                          # group by dimension
      , 'Inspection type'[Inspection item type'   # on the polymorphic resolving column
      , "Inspection count"
      , DISTINCTCOUNTNOBLANK('Inspection'[Inspection item SK])  # aggregating fact per segment
      )
return sumx(inspection_per_type, [Inspection count]) # adds up the result from each segment
```

If there are any sub-selection necessary, KEEPFILTERS can be applied to SUMMARIZE by wrapping it inside CALCULATETABLE.

New data engineers often tackle this sort of problem by creating an artificial single column, such as a concatenation of the item type name and the item SK, and applying a distinct count on that composite column. The approach above avoids any model change.

As with many complex DAX challenges, the idea is to treat it almost as if it were a SQL problem and translate the solution to DAX.

Hidden grain

A generalisation of the previous problem is the case of hidden grain. Power BI supports greater flexibility in DAX and has a more relaxed requirement on table grain compared to traditional dimensional modelling. This allows a single fact table to work at the details with the embedded header rather than having one for header and one for details. Or even present multiple subprocesses of incompatible granularity as one fact table. This is explored further in the chapter *Anticipating questions*.

In these scenarios, the model may have granularity embedded through denormalisation, and not explicitly available as its own grain. In this case, some important aggregating measures such as SUMX, AVERAGEX, and MEDIANX no longer work.

ID dimensions provide a ready solution. They preserve the primary keys of a business process and allow reconstruction of information at the grain defined by that key.

s 22(1)(a)(ii)

The implementation is to use SUMMARIZE on the ID dimension, either directly or through its storage on the fact table as a relationship column, selecting any relevant columns to reconstruct the grain into a temporary table, and then performing aggregation on that table.

Following the previous example of traveller inspection, there is a record of the total inspection time for the traveller which includes the inspection of the traveller as well as any bags on person. Suppose this is recorded as [Inspection duration (mins)] and [Traveller SK] and have been denormalised into the *Inspection* fact table. A measure that calculates [Median time to inspect a traveller] can do so by re-extracting the grain through SUMMARIZE:

```
var inspection_time_per_traveller = SUMMARIZE(
                    'Inspection'
                    , 'Inspection'[Traveller SK]
                    , 'Inspection'[Inspection duration (mins)]
) # reconstruct the table at the traveller grain
Return MEDIANX(inspection_time_per_traveller, [Inspection duration (mins)])
```

In SQL terms, this is similar to using "select distinct" on a set of columns in a denormalised table to retrieve the desired grain, and then aggregating over the result set. The selection of a primary key ensures the resulting grain is correct.

Unsupported relationships

Unsupported relationships refer to scenario when the default filtering configuration of using single-direction, dimension to fact relationships does not easily support a query. Rather than making large structural changes to the model, the data engineer may approach the challenge through DAX.

The most powerful tools for writing queries not naturally supported by the model's relationships are CROSSFILTER, TREATAS, and USERELATIONSHIP.

As seen in *Aggregating dimension values* in *Filtering behaviour*, CROSSFILTER can be used to temporarily turn a relationship to bidirectional. This allows the retrieval of dimension values by another dimension.

TREATAS allows the data engineer to work as if a relationship has been defined, even when none exists. This is particularly powerful in a model with ID dimensions that allow the lookup of any information at the unit-record level. There is a performance penalty for this sort of usage, but it can be the right tool for the correct situation, especially under tight timeframes where adding new relationships is not possible. TREATAS can still be used as long as the information is in the model.

If two tables have two relationships between them where one is active and one is inactive, then USERELATIONSHIP can activate the inactive one and use that as the filtering path. If both relationships are active, one will have precedence, and USERELATIONSHIP can be used to deactivate the primary one. This usage implies that the data engineer has consciously built an additional relationship to support specific measures that use a relationship path different from the model's default. An example is described in *Dynamic type I* in *Filtering behaviour*.

Measure of measures

In a complex organisation, there are always multiple business processes, and multiple measures per process. Quite often, there is an underlying structure to these measures.

Consider a company that handles its own manufacturing, orders, and shipping. These are three business processes, each represented by a measurable fact table: *Manufacture*, *Orders*, and *Shipping*.

Each process has three metrics for operational efficiency:

1. Process volume
2. Median time to process
3. Volume processed on schedule

This results in nine base measures:

1. [Manufacture process volume]
2. [Manufacture median time to process (days)]
3. [Manufacture volume processed on schedule]
4. [Orders process volume]
5. [Orders median time to process (days)]
6. [Orders volume processed on schedule]
7. [Shipping process volume]
8. [Shipping median time to process (days)]
9. [Shipping volume processed on schedule]

These are **base measures** and is one of the types of measures in *Designing measures*.

From these, two **derived measures** are needed:

1. Percentage processed on schedule
2. National comparison

Calculating percentage processed on schedule for each process stage leads to 3 additional measures and 12 total. If national versions are created for all 12, the total becomes 24. This is manageable, but there are two problems with this approach.

s 22(1)(a)(ii)

- **Unarticulated structure.** The measures clearly have a structure to them, but this is not expressed formally. They exist only in the naming convention of the measures.
- **Complex DAX maintenance**. Even a simple set up of three processes lead to 24 measures just on basic measures and not including other metrics such as quality control or profitability. This could easily overwhelm.

Pattern for a measure of measures

The **measure of measures** pattern introduces structure to the measures. Building it requires three elements:

1. A measure table
2. Switch measure
3. Formatting calculation group

The measure table is simply a list of measure names, annotated by columns that describe the measure. Continuing with the example, a table called *'Operational metric'* with three columns:

4. [Measure name] – the names of all the nine measures
5. [Process stage] – "Manufacture", "Orders", and "Sales"
6. [Metric] – "Process volume", "Time to process", "Volume processed on schedule"

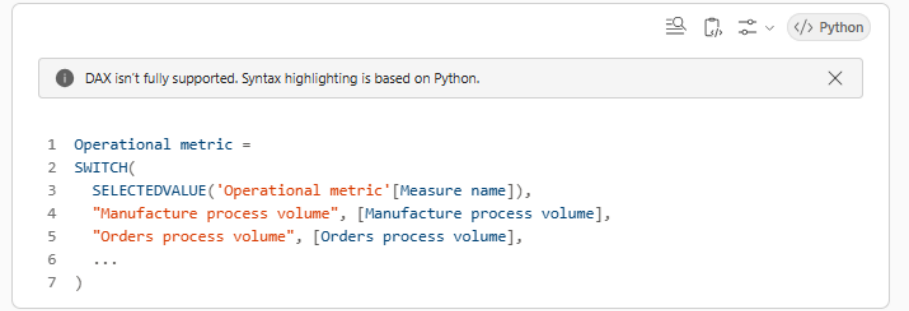
The full table is thus nine rows, one for each measure:

Measure name	Process stage	Metric
Manufacture process volume	Manufacture	Process volume
Manufacture median time to process (days)	Manufacture	Time to process
Manufacture volume processed on schedule	Manufacture	Volume processed on schedule
Orders process volume	Orders	Process volume
Orders median time to process (days)	Orders	Time to process
Orders volume processed on schedule	Orders	Volume processed on schedule
Shipping process volume	Shipping	Process volume
Shipping median time to process (days)	Shipping	Time to process
Shipping volume processed on schedule	Shipping	Volume processed on schedule

In defining this table, it is important to apply a display order where there is a rank. For example, [Process stage] is in order of manufacture, orders, and shipping. It happens these are already ordered correctly alphabetically and the [Process stage display order]

is not necessary in this case. In general, such ranking columns are necessary to express the business order.

The switch measure returns the correct base measure based on the selected row in '*Operational metric*':



This is a **switch measure** because it uses a switch to decide on the calculation. And since the switch is simply to select the value of another measure, it is thus a **measure of measures**.

After defining [Operational metric], three business metrics can be defined by filtering [Operational metric] on the [Metric] value:

```
[Process volume] =
CALCULATE(
    [Operational metric],
    KEEPFILTERS('Operational metric'[Metric] = "Process volume")
)

[Time to process] =
CALCULATE(
    [Operational metric],
    KEEPFILTERS('Operational metric'[Metric] = "Time to process")
)

[Volume processed on schedule] =
CALCULATE(
    [Operational metric],
    KEEPFILTERS('Operational metric'[Metric] = "Volume processed on schedule")
)
```

Derived measures can now be written once rather than 3 times:

```
Percentage processed on schedule =
DIVIDE([Volume processed on schedule], [Process volume])
```

National comparisons can also be written once:

```
National operational metric =
CALCULATE([Operational metric], REMOVEFILTERS('Region'))
```

In this case, the nine base measures are **first order measures**, while [Operational metric], [Process volume], [Percentage processed on schedule], [National operational metric] are **second order measures** because they are calculated base off the switching of the first order measures.

The final element is a formatting calculation group. Power BI does not support dynamic formatting in switch measures by default. Consequently, all values returned by the switch measure do not honour the original format. A calculation group can reapply the format with the appropriate formatStringDefinition:

switch(selectedvalue('Measure'[Measure name]),

“Manufacture process volume”, ‘TODO’,

“Orders process volume”, ‘TODO’,

“Shipping process volume”, ‘TODO’,

....

, selectedmeasureformatstring())

Benefits

The measure of measures has important implications for measure management and report building. In some cases, it is the only viable solution without creating a tangle of DAX definitions or nightmare reports.

The first benefit is simplified DAX management. Instead of repeating near-identical logic across multiple measures, such as three versions of percentage processed on schedule or twelve national metrics, the measure of measures allows a single point of definition.

The second benefit is explicit structure. The underlying logic of the nine base measures is now expressed explicitly in a single table. This enables visuals that are otherwise impossible. For example, a matrix with [Metric] as rows, [Process stage] as columns, and [Operational metric] as values:

	Manufacture	**Order**	**Shipping**
Time to complete			
Process volume			

Volume completed on time			
Percentage processed on schedule			

The formatting calculation group can be used here to ensure all the measure values are presented in the correct format – whole numbers, decimals, percentages.

This visual is impossible to build with 12 separate measures.

The third benefit is powerful reporting. A matrix with [Process stage], [Metric] as rows, [Reporting year] as columns, and [Operational metric] as values allows immediate comparison across years, with conditional formatting to highlight deterioration:

		2022	**2023**	**2024**
Manufacture	**Time to complete**			
	Process volume			
	Volume completed on time			
	Percentage processed on schedule			
Orders	**Time to complete**			
	Process volume			
	Volume completed on time			
	Percentage processed on schedule			

Shipping	**Time to complete**			
	Process volume			
	Volume completed on time			
	Percentage processed on schedule			

This allows the user to see the changes of all measures changing across years in a single visual, with conditional formatting to highlight any significant deterioration.

There are many ways to use the structure expressed by *'Operational metric*'. The [Process stage] column can be used for small multiples in line charts, creating a series of visuals in a controlled and automated manner. It can also serve as a legend to compare metrics directly, such as identifying lags or discrepancies in [Process volume] between manufacture, orders, and shipping.

If each process is managed by a different team, [Process stage] can be used as a report page filter to tailor the view to each manager's concern.

Used appropriately, the measure of measures enables powerful reports that would otherwise be impractical or impossible.

Finally, the measure of measures allows the users to perform operations on the measures rather than the fact tables behind the measures. Inexperienced data engineers attempting the similar outcomes often would try to tackle the problem by forcing the fact table rows into an awkward union. The measure of measures avoids this. In the perspective of *Conforming systems*, the measure of measures allow the data engineer to use the horizontal integration pattern when the vertical integration pattern is not appropriate.

Dangers

Data engineers should use a measure of measures judiciously.

The SWITCH function can be suboptimal when filtering on a column other than the switch value, such as [Metric] instead of [Measure name], or a conformed dimension off [Process stage]. With 40 or 50 branches, performance degrades noticeably. The pattern should be tested in practice for each model.

Dynamism can be overused. In theory, any list of measures can be placed into a switch, but this is not always helpful. The criterion is business expressiveness. The pattern is appropriate when the measures form a coherent structure such as process stages, business lines, or quality criteria in a product line. The pattern is not appropriate when the measures are unrelated or when the structure is unclear.

Report builders can also get carried away. Dynamism is often a sign of bad design. If users must click before seeing what is important, the report is likely poor. A report with ten selectable metrics is not intuitive. The measure of measures should be used to create explicitly expressive visuals, not to hide information behind implicit interaction.

Finally, the measure of measures requires the user to select enough of the measure table's primary key columns to resolve to a specific measure. Without this, the switch returns no value. For example, selecting [Metric] = "Process volume" without specifying [Process stage] will not return a result. This is by design. The pattern is intentionally built **not to aggregate across measures by default**. It prevents accidental errors such as summing what should not be summed or averaging averages. If aggregation across measures is needed, such as summing [Process volume] across all stages or computing a weighted average of [Median time to process], then it must be explicitly defined and tailored to the metric.

Conclusions

Most discussions of the SWITCH function in Power BI focus on its technical role — selecting between multiple measures based on user input, typically via a disconnected slicer table. This is often described as a UI convenience or a way to reduce visual clutter.

However, the measure of measures is not about UI convenience but is about business meaning. Instead of using SWITCH merely to toggle between measures, it is used here to express the structure of the measures themselves. The measure table is a business dimension that categorises measures by meaningful attributes like process stage, metric type, or quality indicator. Consequently, the measure of measures serves, not merely as a technical trick, but a modelling technique that:

- Makes the business metrics, as implemented by the measures, explicit and navigable
- Enables structured reporting (e.g. matrix visuals by metric and process)
- Supports derived measures (e.g. percentages, comparisons) without duplication

In short, the measure of measures is another application of the principles of expressiveness and fragment modelling. Without the anchor in expressiveness, the data engineer is at risk of creating dynamism for its own sake and creating a frustrating report for the user.

Finally, the pattern highlights the philosophy of a good Power BI dimensional model that measures are the face of fact tables, and can be directly manipulated rather than reshaping the facts. The pattern is a natural fit within the horizontal integration pattern from *Conforming systems*.

Row level security

Row level security (RLS) in Power BI means exposing to users only the rows they are allowed to see. This filtering is implemented through a hidden DAX expression based on the user's security role. The filter is applied at a layer above the model—so even the model itself is unaware that the filter has been applied.

Limiting data is the easy part. The challenge is not in hiding sensitive information, but in exposing population-level context for comparison without leaking sensitive details. For example, enabling a regional team to compare its sales performance against national figures—filterable by product type, sales period, and other dimensions—without inadvertently revealing sensitive data from the national population.

This leads to two distinct tasks in implementing RLS:

- Limiting what users can see
- Showing population context

Limiting what the users can see

Power BI implements RLS by applying a DAX expression to the semantic model, based on the user's membership in a role group. The expression can be simple or arbitrarily complex.

There are two main techniques for ensuring different users see different information: Multiple RLS group and RLS group with dynamic logic.

Multiple RLS groups

This approach is straightforward. The administrator creates a separate role group for each user category. For example, one group is created per continent. Each group is assigned a DAX filter that restricts access to the relevant rows. A global team can be assigned to a group with no filter, allowing full access. A recommended name for this group is "*Unlimited*".

This method becomes difficult to maintain if the number of user categories grows. In such cases, a dynamic approach is preferable.

RLS group with dynamic logic

The identity of the user is available via the DAX function USERPRINCIPALNAME(). This allows filtering the model using a mapping table that links each user to the rows they are permitted to see. The mapping table behaves as a many-valued dimension.

This table can be as complex as needed. It may map users to teams, teams to data rows, or supervisors to their direct and indirect reports. Ultimately, it maps the user principal name to a row identifier in the model.

From a performance perspective, the filter should be applied as close as possible to the sensitive data—ideally directly on the fact tables. If the sensitive data resides in

dimensions, the filter can be applied via a snowflake relationship. For example, if the model includes a *Customer* table and the rule is that customers see only their own data.

Multiple copies of the user table may be needed if filtering applies across multiple dimensions. In such cases, care must be taken to avoid conflicting filters, which would result in an unintended “AND” condition.

As with multiple RLS groups, unrestricted users can be placed in a group with no DAX filter. A clear naming convention is helpful—*Limited* and *Unlimited* are preferable to *Restricted* and *Unrestricted*, which can be ambiguous because “restricted” can mean access to restricted data, or that the user is restricted from accessing data.

The dynamic approach pushes security logic into the data layer, allowing transparent, auditable control. However, it shifts administration to managing the access table, which may be populated manually or via automated jobs. The database administration must have adequate controls that are commensurate with the security needs.

If there are complex mapping logic, the user mapping table should be clearly annotated with metadata to explain the assignment in business language.

Showing population context

The challenge with RLS is not just hiding sensitive data—it’s enabling comparison with the wider population without leaking sensitive information. For example, allowing a regional team to compare its sales performance against national figures, filterable by product type, sales period, and other dimensions, without inadvertently exposing sensitive details from the national population.

There are two approaches to solving this problem: the **anonymous approach**, which duplicates the facts, and the **pseudonymous approach**, which doubles the dimensions.

Anonymous (duplicating the facts)

The simplest and safest way to retain population data in the model without risk of leaking sensitive information is to create copies of each fact table, removing all columns that contain sensitive information. If the sensitive information is linked via relationships to dimensions, those relationships should be removed. Without these links, the new fact tables are exempt from the RLS DAX filter.

s 22(1)(a)(ii)

A naming convention helps. The duplicated tables can be suffixed with 'anonymous'. For example, if the original table is *'Sales'* and it contains team details via a relationship with *'Sales team'* dimension, the duplicate table would be *'Sales anonymous'*, with the relationship removed.

All measures that refer to the original fact tables must be duplicated. For example, if *[Sales volume]* refers to *'Sales'*, a new measure *[Sales volume anonymous]* would use *'Sales anonymous'* instead. The formula remains the same but the table reference changes.

This duplication allows safe comparisons. For example:

[Regional sales proportion] = [Sales volume] / [Sales volume anonymous]

Every interaction, including dimensional filters, will work as expected. When a user selects a particular product or time period, the measure will behave correctly without linking to any sensitive information.

When there are many measure pairs, maintaining derived measures can become tedious. In this case, the measure of measures pattern can simplify the measure management. That is, create a switch measure for user data and another for population data. For example, they can be [Regional measure] and [National measure] respectively. All derivatives or comparison measures can be defined on the two switch measures rather than original ones.

Pseudonymous (double the dimensions)

The pseudonymous approach assumes sensitive information is contained in dimensions. For example, the *'Sales team'* dimension may include team member names and locations. In this approach, dimension rows are duplicated via a union, and identifying values are replaced with placeholders such as "(masked)". A binary column [Is masked] is added to indicate whether the row is masked.

The relationship is converted from one-to-many to many-to-many on the original primary key of the '*Sales team'* dimension. The RLS rule then filters users to either the masked or unmasked rows. In the simplest case, where a user either sees all sensitive information or none, only two RLS groups—*Limited* and *Unlimited*—are needed.

Once defined, the model behaves as normal without additional re-work. Users see all information as expected, but sensitive values are replaced with masked placeholders.

Care must be taken when masking. In some cases, the primary key itself is sensitive. For example, if the primary key is a social security number. In such cases, the business key must be replaced with a surrogate key.

Comparison of approaches

The **anonymous** approach is safe and straightforward. It cleanly separates sensitive data by duplicating the fact tables and removing sensitive columns and relationships. This ensures that population-level data is available for comparison without risk of leakage. However, it doubles the model size and increases load time. Maintaining parallel measures can also become tedious, especially when there are many user–

population pairs. In such cases, the measure of measures pattern with switch measures like [Regional measure] and [National measure] can simplify maintenance.

The **pseudonymous** approach is lightweight and preserves interactivity. It duplicates dimension rows and masks sensitive values, allowing the same model structure to serve both restricted and unrestricted users. This approach avoids duplicating fact tables and keeps the model compact. However, because the original primary key is retained, it exposes more detail than the anonymous approach. An informed user may still infer identities from unique combinations of attributes. If the primary key itself is sensitive, it must be replaced with a surrogate key.

Both approaches allow unit-level interactivity. This is intentional to promote the maximum interactivity with all available dimensions. However, this level of granularity allows an educated user may deduce identities from distinctive patterns. For example, “There’s only one team in the country that sells this product at this volume, so this record must pertain to that team.” If this risk is unacceptable, the anonymous approach remains viable but requires additional aggregation to obscure identifying details.

The choice between the anonymous and pseudonymous approach depends on the sensitivity of the data and the risk profile of the user base.

The anonymous approach is appropriate when the data contains highly sensitive information where the risk of inference or reverse engineering is unacceptable. The pseudonymous approach can be used when the access requirements are simple, and the risk of intentional, malicious reverse engineering is low.

This decision is not purely technical—it reflects the organisation’s appetite for risk, its governance requirements, and the expected behaviour of its users.

s 22(1)(a)(ii)

Anticipating questions

One of the expectations for a good dimensional model in Power BI is to anticipate all the business questions that would reasonably be asked during its lifetime, rather than answering the questions that come from defined requirements.

This expectation can take data engineers by surprise, especially those who have come from an IT development background. In IT development of applications, the first step is to clearly define the requirements with stakeholders and build accordingly. While modern development has placed an increased emphasis on iterative refinement and flexibility in responding to requirements, it is still focused on delivering to stakeholders' requirements.

This is not the place to question whether the approach still has merits in today's competitive world of IT. However, it clearly does not work for data analytics. In the world of digital systems, building a form does not lead to another form. No one wants more forms. However, in the world of data, answering one question leads to another question, and no one knows what questions to ask before gaining familiarity with the data. This is part of the fluid nature of data engineering projects.

Consequently, a data engineering project that is focused on building to requirements specifications is destined for frustration. Either the stakeholders have no real interest, or interested stakeholders will continually go back and forth asking for more changes as the project evolves and more data is surfaced.

How does a data engineer anticipate questions without being told what they are? This is much easier than it sounds. Power BI dimensional models are well suited to anticipating questions. Instead of focusing on ticking off stated requirements, the data engineer should focus on the blocks of information captured in business processes, present them as dimensions in a dimensional model, and paying attention to what information are shared by multiple processes. In addition, the data engineer should lean forward and proactively ask what other information would be relevant to the business question at hand, and experiment with adding them.

To develop a model that anticipates questions and captures the breadth of information, the data engineer can remember a simple three-step process:

1. All the facts – business processes
2. All the dimensions – business information
3. All the relationships – mirror their real-world relationship

Each of the step has an artefact to help the data engineer and business stakeholders to ensure the comprehensiveness of the capture.

All the facts

The first step to building a dimensional model that can anticipate all questions is to ensure it captures all the relevant business processes. For ease of memory, this is the step to capture “all the facts.”

This step can be non-trivial for two reasons.

First, in a complex business, many teams work together to complete a single goal. These teams often operate across different business processes. The data engineer and project team are usually working with a particular stakeholder group that focuses on specific processes rather than the full set. For example, some may focus on manufacturing, some on ordering, and others on quality control.

Second, different stakeholders often take different views of the same process. Operational staff may focus on detailed steps, while others take a broader, strategic view. Speaking with any one group always results in a skewed perspective.

In other words, the data engineer faces two challenges — how much to include, and how zoomed out the view should be.

Consider the following example of business processes:

1. Research and development
2. Sourcing materials
3. Manufacture
4. Quality control
5. Orders
6. Shipping
7. Customer feedback

For the question “what are the different business processes to include?”, the data engineer should be as comprehensive as possible during horizon scanning. This means being persistent in asking “where does this come from?” and “where is this going to?” until the end-to-end is covered. A useful rule of thumb is that a business process should start from outside the organisation and leave outside again.

Once all the processes are identified, it becomes easier for stakeholders to collectively decide which ones to include. Usually, there is a shared sense of what is core and what is peripheral. This is a different question from “what are your requirements?” Requirements tend to narrow the perspective.

For example, a manager of quality control may focus narrowly on their part of the process. If the project is driven by this stakeholder — perhaps because quality control

initiated the engagement — the requirements may be framed as “how much is spent on quality control?” or “how many products are failing?” Such a view is unhelpfully narrow. Instead, the data engineer can help by identifying the full business process. The manager may then recognise that, even though reporting on the cost of manufacture is not a “must-have requirement” for quality control, it is still a core business process. Research and development, by contrast, may be considered peripheral.

For the question “how zoomed out should the view be?”, the answer is to work at a level that makes sense for decision-making. It does not help to model the lowest-level details such as sending emails or looking up documents. On the other hand, it is also not helpful to see manufacturing and quality control as one large step.

Zooming out often involves grouping sub-processes into a single process, or denormalising detail rows into header-level structures which would expand the grain. For example, the quality control process may involve multiple testing steps for different criteria. Rather than creating one fact table for quality control and another for its testing steps, it is more expressive to place them under a single fact called ‘*Quality control*’ rather than two for ‘*Quality sample*’, and ‘*Quality sample test*’ etc.

s 22(1)(a)(ii)

Power BI’s flexibility with its DAX engine supports this approach. In simple cases, a DISTINCTCOUNT can recover the embedded grain, such as counting headers by their ID. In more complex cases, DAX can still retrieve the hidden grain, as explained in *Designing measures*.

This question often poses a challenge for engineers trained in traditional dimensional modelling and learning Power BI for the first time. The classical approach tends to split facts by grain rather than by business process. This can obscure the business view.

The artefact for this step is the **linear process diagram**. A linear process diagram is a business process diagram using only linear steps with no branching nor cycles nor decision trees. Readers find linear diagrams much easier to understand than diagrams with arrows pointing in all directions. The constraint of linearity forces the business analyst to abstract the process to a level that is useful for themselves and others. This is especially true for executive audiences.

It can also be helpful to categorise the business processes into higher-level categories. Continuing with the example above:

- Design – Research and development
- Production – Sourcing, Manufacturing, Quality control
- Sales – Orders, Shipping, Customer feedback

[DIAGRAM

Design | Production | Sales

R&D -> Sourcing -> Manufacture -> QC -> Orders -> Shipping -> Feedback

]

Making linear process diagrams does not mean that business analysts cannot create more complex diagrams for other purposes. However, these are not linear process diagrams for model planning.

The identified business processes form the facts of the dimensional model. Hence this step can be remembered as “all the facts.” The processes selected for the model define its scope. For example, if the decision is to focus on manufacturing onwards, then questions about research and development or sourcing will fall outside the model’s scope.

Identifying the business processes is the hardest part of anticipating questions. It involves making business decisions and requires a high degree of judgement and practical experience. Once the processes are identified, the next step of identifying the business information follows more mechanically.

All the dimensions

The second step, after identifying the business processes in scope, is to identify all the business information known to those processes. This ensures that all questions about the processes included in the model can be answered. For ease of memory, this is the step to capture “all the dimensions.”

This step is more straightforward than identifying the processes. Business stakeholders usually know what information is captured. While there may be debate about the relative importance of one attribute over another, the answer is often that they are all important. If they were not, they would not be captured in the first place.

The nuance for the data engineer is to distinguish between what is captured by a business process and what is known to it. The general rule is that downstream processes inherit information from upstream ones.

Consider the following example. Suppose these are the information captured by each process:

- Manufacturing — Product type, Manufacture details (date, batch number)
- Quality control — Testing details (manufacture sample, date, criteria, results)
- Orders — Product type, Customer, Order details (date, order units, sales amount, order number), Order status
- Shipping — Shipping order (order number, line items), Shipping logistics (shipping method, shipping company), Shipping status (date, status)

- Customer feedback — Customer, Product type, Feedback details (method, star rating, comments)

In this example, the order process captures product type, customer and order units. The shipping process does not capture these directly but knows them through the order number. This is inheritance. However, not all downstream processes inherit everything. For example, customer feedback may not record the order number and may only capture the product type. In that case, it does not know the order date, even though it is downstream.

Once the information is identified, it can be presented using a cumulative information diagram. This is a table with business information as rows and business processes as columns, both listed in chronological order. The diagram is cumulative because information typically accumulates downstream through inheritance.

It is also helpful to annotate the diagram with where the information is physically stored — which database or table holds each attribute. This helps assess what can be reasonably answered and informs project planning.

The diagram for the example is below to convey what are the information known to each business process. “C” is used to indicate capture, and “I” is used to indicate inheritance of information.

Thus, the linear process diagram is a description of the real-world business processes.

	Manufacture	**Quality control**	**Orders**	**Shipping**	**Customer feedback**
Product type	C	I	C	I	C
Manufacturing details	C	I			
Testing details		C			
Order status			C	I	
Order details			C	I	
Customer			C	I	C
Shipping order				C	
Shipping logistics				C	
Shipping status				C	
Feedback details					C
Database storage	XYZ SQL database	ABC diagnostics database	XYZ SAP ERP	XYZ SAP ERP	Salesforce analytics

While the cumulative information diagram builds on top of the linear process diagram by drawing a parallel diagram of how information flows through the business process, and the storage databases, or database tables.

In other words, the cumulative information diagram maps the real-world processes to the database-world.

Since business information becomes dimensions in a dimensional model, this step can be remembered as “all the dimensions.”

All the relationships

The final step is to ensure that the business processes and their known information are reflected in the dimensional model through filtering relationships. If all relevant processes are captured, all relevant information is identified, and valid relationships are correctly implemented, then the model can truly anticipate all questions.

The rule of thumb is simple. If a business process knows a piece of information, whether by direct capture or by inheritance, then the corresponding dimension should filter that fact table.

Continuing with the earlier example, suppose *'Order ID'*, *'Order calendar'*, and *'Customer'* are dimensions, and *'Order'* is the fact table. These dimensions should directly filter *'Order'*. The *'Shipping'* fact table should also be filtered by *'Order ID'*, since shipping needs to know the order number. In this case, *'Order ID'* is a conformed ID dimension for both processes. The *'Order calendar'* and *'Customer'* dimensions should also filter *'Shipping'*, because this process inherits them through the order number.

New data engineers often forget about inheritance. This creates a frustrating experience for users who expect to retrieve all shipments for a customer but find the model does not support the question due to a missing relationship.

There are exceptions to inheritance. Sometimes an upstream process has a zero-to-many relationship with a downstream one. In these cases, including the upstream information as a filter may be problematic.

For example, suppose the quality control process usually tests manufactured batches, but sometimes tests other aspects not tied to a specific batch, such as testing factory equipment itself. In this case, *'Manufacture ID'* should filter *'Quality control'* so users can look up testing for a batch. However, it may be better not to implement a filtering relationship between *'Manufacture calendar'* and *'Quality control'*. In Power BI, if a user selects a manufacture date, this relationship will also filter *'Quality control'* and affect any measures, even when the test was unrelated to a batch. It may be sufficient to use a role-playing *'Reporting calendar'* that filters *'Manufacture'* on manufacture date and *'Quality control'* on testing date. This avoids misleading results. If the relationship is implemented, the report design must be careful not to mislead users with incorrect numbers caused by unintended filters.

Usually, upstream processes tend not to know about downstream processes. This means that *'Customer'* should not filter *'Manufacture'* because this information does

not exist at the time of manufacture. However, the data engineer has the benefit of hindsight so that some information can be brought back to the earlier process. For example, while the results of quality control of a manufacture batch are not known at the time of manufacture, the result can be associated to the batch number afterwards. To express this back at the batch number grain, multiple quality control criteria will need rolling back to the batch for an overall pass or fail. A dimension for *'Batch quality outcome'* can be used to filter '*Manufacture'*. In general, storytelling dimensions summarise the journey of an entity through the whole business process and can be associated to all steps of the process in hindsight.

s 22(1)(a)(ii)

The artefact for this step is the **chronological bus.** A bus matrix is a table that lists dimensions as rows and facts as columns, showing which dimensions filter which facts. In Power BI, this should reflect the actual filtering relationships, including their cardinality. A chronological bus matrix is one where the facts and dimensions are sorted in chronological order of their occurrence in the business.

The diagram for our example is below. The symbol 1 -> * is used to indicate one-to-many, while * -> *, which is used for product type and customer feedback, indicates a many-to-many relationship. The latter would mean that one customer feedback may be on several products at once.

	Manufacture	Quality control	Orders	Shipping	Customer feedback
Reporting calendar	1 -> *	1 -> *	1 -> *	1 -> *	1 -> *
Manufacture ID	1 -> *	1 -> *	1 -> *	1 -> *	
Manufacture calendar	1 -> *		1 -> *	1 -> *	
Batch number	1 -> *	1 -> *	1 -> *	1 -> *	
Batch quality outcome	1 -> *	1 -> *	1 -> *	1 -> *	
Product type	1 -> *	1 -> *	1 -> *	1 -> *	* -> *
Testing ID		1 -> *			
Testing date		1 -> *			
Quality criteria		1 -> *			
Testing result		1 -> *			
Order ID			1 -> *	1 -> *	
Order calendar			1 -> *	1 -> *	
Customer			1 -> *	1 -> *	1 -> *
Order status			1 -> *	1 -> *	
Shipping calendar				1 -> *	
Shipping company				1 -> *	
Shipping status				1 -> *	
Arrival date				1 -> *	
Feedback calendar					1 -> *
Feedback sentiment (star ratings)					1 -> *

The chronological bus can be used to evaluate what questions are answered by the model.

In the above example, because '*Customer'* is a conformed dimension for *'Orders', 'Shipping',* and '*Customer feedback'*, it is possible to identify the customers by the volume of orders, shipping delays, and feedback results.

Because '*Product type'* is a conformed dimension on all processes, it is the attribute that the model can answer most questions for, such as the results of quality control versus the customer feedback. Note, however, the many-to-many relationship between '*Product type'* and '*Customer feedback'* indicates that any assignment of feedback to product is approximate and may be double counting.

Because '*Manufacture ID'* is a conformed ID dimension from *'Manufacture'* to '*Shipping'*, it means that the model can support traceback of faulty products that have been shipped.

On the other hand, it is not possible to analyse the effect of shipping on customer feedback because the relationship between the dimension and fact does not exist. Such analysis may be possible indirectly through the '*Reporting calendar'*, '*Product type'* and '*Customer'*, which together may narrow the transactions down to a segment that correlates shipping status and shipping company on feedback sentiment. If this can be done systematically, it can be introduced as a pre-computed information in the data pipeline.

Ensuring that the model's relationships match the real-world flow of information means the model behaves in ways users intuitively expect. Downstream processes tend to inherit from upstream ones, and except for summary tables, upstream processes never know about downstream ones. As a result, the chronological bus is typically dense in the upper right. This structure makes gaps easy to spot. These gaps may be accidental, reflect a limitation in the source system, or be a deliberate design choice.

In the example above, '*Manufacture calendar'* does not filter '*Quality control'*. This is a design decision. It does not filter '*Customer feedback'* either, which is a limitation because the information is not recorded. This example does not have any accidental gaps. If there were, they would stand out.

The chronological bus only works if the data engineer has modelled information as dimensions, processes as facts, and listed them in chronological order. Consider the same matrix but the columns and rows are sorted alphabetically:

	Customer feedback	**Manufacture**	**Orders**	**Quality control**	**Shipping**

Arrival date					1 -> *
Batch number		1 -> *	1 -> *	1 -> *	1 -> *
Batch quality outcome		1 -> *	1 -> *	1 -> *	1 -> *
Customer	1 -> *		1 -> *		1 -> *
Feedback calendar	1 -> *				
Feedback sentiment (star ratings)	1 -> *				
Manufacture calendar		1 -> *	1 -> *		1 -> *
Manufacture ID		1 -> *	1 -> *	1 -> *	1 -> *
Order calendar			1 -> *		1 -> *
Order ID			1 -> *		1 -> *
Order status			1 -> *		1 -> *
Product type	* -> *	1 -> *	1 -> *	1 -> *	1 -> *
Quality criteria				1 -> *	
Reporting calendar	1 -> *	1 -> *	1 -> *	1 -> *	1 -> *
Shipping calendar					1 -> *
Shipping company					1 -> *
Shipping status					1 -> *
Testing date				1 -> *	
Testing ID				1 -> *	
Testing result				1 -> *	

In comparison, this matrix is difficult to read. This is the case even though the dimension names have already been purposely prefixed to group similar attributes together, and that “Manufacture”, “Order” and “Shipping” happen to sort chronologically. In any other instance, the bus matrix would be incomprehensible.

Consequently, the three-step process of linear process diagram, cumulative information diagram, and the chronological bus is not an accident. It is designed to make it simple for the data engineer to keep track of information, visualise what questions can be answered, and thus accomplish the goal of anticipating questions.

The bus matrix is also a chance to check naming. Dimensions should be nouns because they represent business attributes or information. Facts should be verbal because they represent business processes.

This matrix should be generated rather than created manually. See the chapter on *Automation*.

The relationship between business information and business processes is mirrored in the relationship between dimensions and facts. Hence this step can be remembered as "all the relationships."

Conclusion

The three-step approach guides the data engineer to work on the information captured in business processes, rather than the stated requirements of "I would like to see X by Z". The focus is on preserving information, not on answering specific questions.

The three steps and their artefacts can be summarised in a table like thus:

Step	**Content**	**Artefact**
All the facts	Identify all the business processes.	**Linear process diagram.** Shows all the major steps in a business, set at a level that makes sense to decision-making.
All the dimensions	Identify all the information captured by the processes.	**Cumulative information diagram**. Shows all the information relevant to the business, how they flow, and where they are physically stored. Used to inform what can be reasonably answered.
All the relationships	Relate information to their processes in the dimensional model.	**Chronological bus**. Shows the dimensional model that is implemented. Used to identify what possible questions that can or cannot be answered, particularly any gaps.

Following these three steps ensure the full range of information is preserved, and questions relating to this information can be answered because the relationships exist in the model.

This breadth of information requires a variety of techniques to integrate effectively. The technical patterns introduced in earlier chapters are tools the data engineer can use to meet these diverse scenarios. In particular, the inclusion of ID dimensions guarantees that every grain can be recovered when needed. Storytelling dimensions, on the other

hand, support queries at the grain of the entity of interest, allowing users to answer questions at that level.

In future, when the user asks a different question, if the information is captured in the model, it can be answered because it has already been articulated in the dimensional model. This is how the data engineer anticipates questions.

Quality & reliability

The foundations of trust

As emphasised throughout, the data engineer's attitude must go beyond "garbage-in, garbage-out." The role is not to passively present data from source systems to business users. The role is to value-add.

Some of the most powerful ways to add value are covered in the *Storytelling* chapter. This section expands on how the data engineer can add value to quality.

Recall that the data world is an imperfect projection of the business world into the database world. The data engineer's task is to reshape the data world so that it better mirrors business intent and becomes useful for decision-making.

Data quality issues refer to the gap between the business world and its projection onto the data world. For example, key information required for decisions may not be captured, or may be captured in free-text format. When business users encounter these gaps, they experience a mismatch between the product and the business world they already know. In many cases, this means they cannot trust the product for decision-making. No amount of blaming the digital system will change this reality. From the business's point of view, the product simply cannot be used.

In this perspective, one of the data engineer's jobs is to "plug the gap" left by the projection — to the satisfaction of the decision-maker. This is how the data engineer adds to data quality.

Beyond the imperfections of the projection itself, the mechanical processes of a data pipeline may introduce further errors. An example is schema changes. Users may experience this when entire tables become empty or when inconsistencies arise between business processes. On some occasions, this can be disastrous. For example, if the cost forecast has been updated to the latest month but the actual expenditure has not. This would result in an unacceptable discrepancy in the final account reconciliation.

The data engineer must build a pipeline that minimises the impact of such errors. This can be loosely called the reliability of the pipeline.

Ultimately, data quality or reliability issues are detrimental to users' trust of the data. Conversely, quality and reliability are foundations of trust.

Quality metadata

Data quality issues reflect the gap between the business world and the data world. In this view, metadata is not merely documentation for the next person working with the data. It is an indispensable tool for bridging that gap by adding the interpretative context that connects the user to the business reality behind the data.

If a column in the source system is called [Valid], renaming it to [Is current name] is not the difference between "having metadata standards" and "not having metadata standards" — it is the difference between comprehensible and incomprehensible data.

This bears stressing. Quality metadata adds value in a way that no other technique can.

Metadata is also the first impression for any user engaging with the data engineer's product. It is the first step to trust. It can determine whether a user engages with the product or walks away.

Creating quality metadata is not difficult. Compared with the technical demands of data engineering, metadata is perhaps the easiest. All it takes is for the data engineer to ask: *"What does this look like to the user? If I were the user, is this something I would want for myself?"*

Importantly, metadata is not an add-on task. It is part of the modelling task itself that starts with exploratory analysis. In complex cases, starting with metadata is a useful step for laying out the logic in plain language before implementation. A mature data engineer will acquire the habit of writing metadata first, not last.

s 22(1)(a)(ii)

This section covers three aspects of metadata:

- Names
- Business descriptions
- Database keys

Names

Each site will have its own naming conventions. This chapter is not about defining a specific naming convention, but on universal principles that should be observed in all naming conventions.

Names apply to schema, tables, columns, Power BI models, measures, calculation groups, and display folders.

In every instance, names should be **business centric** and **plain language**. They must not use idiosyncratic acronyms or abbreviations for the developer's convenience. There

s 22(1)(a)(ii)

are no justifications for using [Employee cnt] when the meaning is [Employee count]. If the infrastructure supports spaces in column names, they should be used.

Names should **contain helpful signals**. For example, “Is” and “Has” can be used to indicate binary columns, such as [Is non-compliant]. Columns ending with “date” and “datetime” should respectively hint at their precision. Tables starting with Ref indicate a reference table. These signals should not come at the cost of readability. This is why a Power BI dimensional model should not prefix tables with “dim” or “fact.”

Naming conventions should be **consistent** across the schema, the database, and the warehouse where possible. Consistency in prefixes, suffixes, casing, and syntax creates a design language for the warehouse. Just as Apple or Google products use consistent icons and layouts to create a seamless experience, a warehouse that speaks a consistent language empowers users.

In consistent naming conventions, the data engineer should pay particular attention to **conforming concepts**. This means that when a data engineer uses a name, it is important to check whether the concept is already named differently and shared by other domains. For example, when adding *Cake.RefQualityControl*, it is important to check whether *RefTesting*, *RefAudit*, or *RefInspection* are being used to mean the same thing. If so, it is worth reusing an existing name.

Consistency applies to all aspects of the name — capitalisation, syntax, semantics. It is better to stick with [Sales date] and [Inspection date] or [Date of sales] and [Date of inspection] and not switch from one to another haphazardly.

Names should be **explicit** and **unambiguous**. An explicit column name means using [Inspection first of month] rather than [First of month]. An unambiguous name means ensuring the word has only one meaning for the user. For example, does [Last inspection date] mean “previous inspection date” or “the final inspection date”? Does “restricted” mean “less” or “privileged”? Special attention should be paid to words that are both nouns and verbs. They are particularly prone to misinterpretation.

An extension of explicit and unambiguous names is for them to be **standalone**. Data tables and columns can often be taken out of context. For key concepts, it could be worthwhile to make them carry their own context. For example, [Is non-compliant] may be better as [Is financially non-compliant]. This may involve repeating the table name into the column name.

Another extension of explicit and unambiguous names is **avoiding generic filler words.** The words “type”, “group”, “summary”, and “details” are often used by developers without any meaning, much like “ahs” and “umms” in a sentence. These tend to obscure rather than clarify. For example, if a table is called “summary”, it should be

renamed to reflect what is being summarised. And “details” should be avoided unless it is truly a finer grain expansion of a header row.

In choosing names, the data engineer faces the tension between explicit accuracy and succinctness. Sometimes an overly verbose name is not aesthetically pleasing in a report visual. When in doubt, the data engineer should err on the side of explicit because it is easier to shorten a name in Power BI visual than for a user to creatively extend it.

These principles apply just as much to code. Temporary tables and CTEs should be named with business meaning, not cryptic abbreviations like *tmp*. In complex code, column renaming should happen early, on first contact, and persist throughout during transformation. This practice promotes clarity and readability. As the Zen of Python says: *Readability counts*.

s 22(1)(a)(ii)

The following is an example of naming conventions for prefixes for inspiration:

- *Earliest / Latest* when referring to timed events or versioning. For example, the earliest activity date and the latest activity date, or the latest version. Do not use *last* when one means *latest*.
- *First / Last* when enumerating sequences such as 1, 2, 3. Beware that *last* can be confused with *previous*. Use *final* if acceptable.
- *Before / After* when emphasising a transition in time. Such as in an audit table of column value changes, use [Before value] and [After value].
- *Previous / Next* when referring to consecutive sequences. In the context of code, not for presentation, *self* can be used for the current context.
- *Primary / Secondary* when emphasising importance. For example, the [Primary supplier] if a shipment of goods has multiple suppliers, or the [Secondary supplier] if there were more than one.
- *Start datetime / End datetime* when referring to periods, such as in slowly changing dimensions or referring to statuses. Avoid having one in a table but not the other.
- *Creation / Update / Delete* when referring to records, such as [Creation date], [Creation user], [Update user], [Delete datetime]. In source systems there are often markcolumns such as [Last updated] and [Last updated by]. These should standardise to [Update datetime], [Update user ID].
- *Current / Historical* when referring to records that have changing validity. Usually, the current record is the one where the end datetime is in the future, such as ‘9999-12-31’, and historical for those end-dated. Do not use *history* to

s 22(1)(a)(ii)

refer generically. For example, do not call a table *Cake.SalesHistory* if it suffices to say *Cake.Sales*. As a rule of thumb, current/historical should be used for mutable records, and not for immutable records.

Business description

Business descriptions are frequently overlooked by data engineers as an add-on rather than a core task. Too often, they merely restate the artefact name — describing *Cake.Sales* as "A list of cakes sold" or [Is non-compliant] as "True if found non-compliant." These descriptions do nothing more than take up storage space. More often, they frustrate users who navigate to the description expecting clarity, only to find something unhelpful. At worst, they harm trust by conveying shallowness.

Instead, business descriptions are **meaning explanations**. They enhance data quality by aligning artefacts to business realities through interpretative context.

Like names, descriptions should be business-centric and written in plain language. But unlike names, they should be substantial. A good description should focus on:

1. Business realities
2. Transformation logic, including assumptions
3. Limitations
4. Linkages and comparisons with related concepts

s 22(1)(a)(ii)

Business realities, rather than technical notes, should be the focus of descriptions. For example, a column like [Is non-compliant] should describe what non-compliance means in business terms, not "if the result code is 'F'."

Data engineers, as well as stakeholders, often slip into using system names as shorthand for business concepts. This is not preferred. Descriptions should refer to business entities and business processes, not the systems that implement them. For example, instead of "Salesforce" or "SAP," use "health care record" or "employee registration."

There will be cases where system names or technical implementation details are necessary for transparency. These should be included as supplementary information — not as standalone descriptions. For example, the description for [Is non-compliant] may be:

"True if the transaction failed to meet the minimum regulatory requirements for importation, based on the department's published compliance criteria. This includes both automatic and manual assessments. Transactions flagged by automated rules are included regardless of whether they were later manually overridden. This column does not include cases where compliance was not assessed. The result code is derived from the ResultCode field in the source system, where 'F' indicates failure."

Transformation logic is the heart of a data pipeline. Describing the transformation logic and any important assumptions is necessary for transparency and trust. This is especially important for complex or fuzzy logic.

The same principles apply. The transformation should be stated in business terms. Technical statements can be included as elaboration, but not as a substitute for meaning.

For example:

“This column is derived from the inspection result and officer notes. It is true if the inspection outcome was ‘Spoiled cake’ or ‘Not tasty’, and the inspection was conducted by a certified officer. It excludes inspections conducted during training. The logic is implemented in the InspectionOutcomeFlag measure using a combination of result codes and officer certification status.”

Transformation logic can include **lineage** from data sources where relevant.

Limitations, should similarly be framed in terms of business interpretation. “The column can have a null value” is not a limitation. A better statement is:

“The inspector may not record a result if there are no issues. This means it is not possible to distinguish between a successful inspection and an inspection not performed.”

Clear statements of limitation are essential for transparency and reliable application of business insight.

Linkages and comparisons are often valuable for users exploring complex business processes. Where relevant, pointing out related concepts in descriptions can be tremendously helpful for users navigating the warehouse. These are akin to the “Related links” in an encyclopaedic website.

In terms of storage, if the same description applies in multiple places, it should be included in the reference table, not the transaction table. Reference tables are the carrier of meaning and the user’s primary interface to the model. Of special importance are storytelling dimensions, which should have the richest and most articulate descriptions to capture the complex logic they encapsulate.

Descriptions on database artefacts — tables, columns, measures — will inevitably be piecemeal. Therefore, standalone documentation is also necessary to provide an end-to-end view of the business problem and its data model.

Database keys

Database keys include primary keys, foreign keys, and unique keys. They, too, are metadata. While names and business description explain artefacts in and of themselves, keys are necessary to explain how they relate to each other.

First, **relationships are part of meaning**. The relationship of a table with another table tells the user something about both tables that neither one can in isolation. For example, a foreign key between *Cake.Sales* and *Cake.RefProfitability* immediately signals the idea that some sales may be profitable and some are not.

Second, database keys help users navigate the database and perform joins. In this sense, database keys serve as a map for the warehouse. Without them, users are left to guess. This is not acceptable.

Third, primary keys play a special role. As explained in *Mapping the data world*, the primary key is what links a database record to its real-world business entity. Therefore, they are necessary to convey the meaning of a row to the business user.

In addition, primary keys are the foundation for fault tolerance and change detection. They allow the data engineer to identify duplicates, track changes, and implement incremental loads. These techniques are covered in the chapters *Fault tolerance* and *Load mechanics*.

There are cases where primary keys are difficult to define. If the answer is “each row is a new entity,” such as in a log table, the data engineer can still express this with an increasing integer and load incrementally. In other cases, the grain of the fact table may be unclear or difficult to compute, especially when business centricity requires compacting several grains into one fact table for presentation. Even so, it is worth the effort of computing a key if possible.

Each organisation implements keys differently. Some store them only in conceptual diagrams or third-party tools. The disadvantage is that they cannot be easily read or queried. The best approach is to house them as standalone tables in the warehouse. This makes them visible and open to querying like every other table in the warehouse. In other words, metadata should be stored and treated as data.

Ultimately, keys are not just technical constraints. They are statements of relationship. This makes them a non-negotiable part of metadata.

Conclusion

Data without metadata is data without context. Consequently, the primary purpose of metadata is not governance, tagging, discoverability, or data exchange. These are all important, but they are secondary effects.

Instead, metadata should be seen in the context of the data engineer's task is to shape data in light of business interest. From this perspective, the primary purpose of metadata is to add interpretative context that aligns the data world to its underlying business realities. Without this context, interpretation is left to the user who must fill in the gap by guesswork. When the user must guess, the data engineer has failed to shape the data to meet business interest.

Quality metadata is not difficult to do. It comes naturally with the habit of looking at the product from the user's perspective, and a taste for the satisfaction that comes from articulate writing.

Finally, whether it be names, descriptions or keys, metadata should be stored and available as data. Instead of recording them in diagrams, specialist tools, or database constructs alone, they need to be available as tables. Treating metadata as data enhances the ability to distribute them, perform automation, or surface to appropriate tools using APIs. This maximises the impact of metadata. Large language models are also adept at reasoning with such structures and putting them to good use. Creating standalone names makes it easier for large language models to reason correctly on top of metadata.

Dealing with data quality

Data quality issues arise from the gap between the data world and the business world. One way to see this gap is as the imperfection left by projecting business processes onto database records. For example, certain events should be recorded only once but are duplicated, or records are missing, or the information recorded does not reflect reality.

In general, errors in the database are biased toward the advantage of the person providing the information. For example, both government social services and the tax office are concerned with underestimation of income. However, the tax office is more likely to record lower estimates because taxpayers are unlikely to complain if they accidentally report less income — they receive a higher tax return. But if they underestimate income for social services, they are more likely to notice the error because they receive less benefit than expected.

Most people see data quality issues as such failures of data capture to reflect reality. However, this is a simplistic view.

A more sophisticated view is to see the gap not between the business process and the data world, but between the data world and the business intent the data engineer is trying to meet.

The premise is that one business's view of "good enough" is another's view of "not good enough." Consider the example of recording personal income. A government social service is more concerned about underestimates, while a bank credit rater is more concerned about overestimates. Both prefer perfect accuracy — but at what cost? At a finite cost, social services actively manage underestimates to meet legislative requirements, while credit raters manage overestimates to reduce risk.

There is no accuracy without margin of error, no margin of error without a risk threshold, and no risk threshold without an articulation of business objective. Consequently, all quality issues are failures of applying business intent.

The first view can be described as "data vs reality" and the latter "data vs business intent".

A dogmatic insistence on "reflecting the real world to the dollar" is costly and impractical. It is driven by an ideology of a technological society and may infringe on other ideologies such as privacy protection. It is not the data engineer's job to promote technological ideology. In most cases, it is an exercise in frustration, in the worst case, a dogmatic insistence on this view can paralyse a project because of perceived imperfections. Instead, the data engineer's job is break through the paralysis in serving business intent.

With that in mind, common data quality issues typically arise because digital systems and business processes were not built to honour the full intent of the business. This could be due to poor design (e.g. missing data type constraints), biases in the business process (e.g. social services vs tax office), or a mismatch of purpose (e.g. import declarations captured for tax but analysed for biosecurity). The data engineer should be equipped with a set of tools to deal with these scenarios. The following examines these techniques through three high-level categorisation – human curation, precise business rules, and fuzzy logic.

One theme across these techniques is the need to monitor the validity of an assumption and trigger an automated alert when it is violated. This approach is described in detail in the chapter *Tests and assumptions*.

Human curation

If data quality issues arise from a gap between the data world and business intent, one of the simplest remedies is to allow business experts to intervene directly.

Data annotation

The most common form of human curation is data annotation. For example, a digital system may record sale locations as-is by store address. A business expert may later annotate these locations into sale regions or company sites. This annotation occurs in the data pipeline, not the source system, and is commonly known as reference data management.

Another example is harmonising employee identities across systems. In large organisations, the same employee may appear under different accounts for different business processes. A business expert that takes the enterprise view of the operations may annotate these accounts to map them to a canonical representation. This is known as master data management.

Human annotation extends beyond reference and master data. It applies wherever a human can interface with records outside the operational context. There are two legitimate views of data: one fast-paced and operational, suited to digital systems; the other reflective and slow-paced, suited to analytical work. Both may require human input. It is not reasonable to expect digital systems, with their modern challenges, to carry the full load of the data work. Instead, human input for an analytical view can take place after the fact, and whenever that is the case, human annotation applies.

Traditionally, data warehousing treats human curation as an exception. But if human editing can be implemented transparently, with control and auditability, it should be treated as a standard business tool. The rise of AI and the need for human-in-the-loop is shifting this view. Rather than bespoke solutions, new tools may emerge to make human curation a first-class citizen in data-driven organisations.

Whether it's reference data, master data, or other forms of annotation, the data engineer must monitor for incoming records that require curation. For example, uncategorised store locations or unmapped employee accounts. This can be done through automation, as described in *Tests and assumptions*, or through a data quality report, covered later.

Applying assumptions

Another direct way for business experts to intervene in data quality issues is to apply business assumptions and act when those assumptions are violated.

Consider the recording of dates. If users manually enter dates, they may accidentally type 2300 instead of 2030. If the digital system lacks validation, such errors can end up in the database. Even a single mistake can distort a line chart or skew time-based analysis.

One way to address this is to assume that future dates must be within 50 years. Dates outside this range are treated as invalid and converted to null. This assumption should be monitored, and if violated, an alert should prompt a business expert to correct the source data.

Another example is assuming uniqueness. If data entries are almost always unique, but occasional duplicates occur, the pipeline can assume uniqueness, ignore duplicates, and monitor for violations. If duplicates appear, the business expert can intervene at the source.

This approach works best when the business expert can correct the issue for the next batch load. It is suitable for rare, irregular violations that do not require bulk remediation. If data quality issues are more frequent or systemic, a data quality report may be more appropriate.

Data quality report

Data quality reports are suitable when issues are frequent, numerous, or easier to treat in bulk on a periodic basis.

An effective implementation is through a combination or choice dimension. A combination dimension may include one column per issue and describe each transaction. For example, a dimension called '*Sales data quality'* might include [Is missing sales amount], [Is invalid sales date], [Is unknown customer], and so on. Coupled with the ID dimension, this dimension makes it easy to build a Power BI report that surfaces transactions requiring remediation by a business expert.

A data quality dimension also supports systematic analysis by enabling statistics on the number and type of issues.

Reports are only effective if they embed in the user's workflow. A de-contextualised report becomes forgotten and unused. The same applies to data quality reports. One way to achieve this is to ensure the business has a workflow trigger that prompts engagement with the report. More on designing data products that embed in workflows is covered in the chapter *Gathering requirements*.

Precise rules

Translating business knowledge into precise rules allow for their automation and reduces the reliance on human intervention. One way to look at these rules is to see them as "fixing issues" in the data, but a more productive way is to see them as rules that help bridge the data with business intent. The following are 4 examples:

1. Defining analytical concepts
2. Defining primary key
3. Defining the primary record
4. Defining relationships

Defining analytical concepts

Since business insight is information analysed in light of business interest, a direct way of improving data quality is to formulate this analysis as a defined concept. This concept becomes a lens through which the data is interpreted. It allows the business users to see the data in a way that is more expressive of business intent.

Good and bad entities

The prime example is that of "good" and "bad" entities. A business process may collect detailed information such as audit results, inspection results, incidents, sentiment, or sales volume. However, these are yet to be formulised into "good" and "bad" in a systematic way. It could be that numeric values such as star ratings need to be translated into good and bad using thresholds. More commonly, it is because a business process records information at a lower level that needs to be rolled up back to the entity of interest. For example, a manufacture batch may have multiple quality control criteria, where failing one critical criterion, or multiple non-critical criteria, will define a failure at the batch level.

It is not reasonable to expect the digital systems to always define these concepts at operational level. Their primary job is to execute workflows while preserving data accurately. The data engineer play the part by adding this analytical lens.

Milestones

Milestones are another important analytical definition. A business process typically has many detailed steps, some of which may loop back. This plethora of steps can confuse and bury insight in a mass of details. A useful analytical view is to define major milestones that can be used to measure performance. To be useful, there should be no more than seven milestones, and more importantly, each milestone should be

associated with a specific control or a responsible owner who can act on the timeliness of reaching that milestone. To deal with loops, the data engineer can define the earliest and the latest time such events occur.

Conformed dimensions

Another form of analytical concept is the creation of conformed dimensions. A large organisation may have many processes, and many share nearly similar concepts under different names. A little effort can create a higher view that reconciles these specific instances. When done appropriately, a conformed view can empower decision makers at the most senior level in the department.

Defining analytical concepts takes leadership and negotiation skills. The data engineer is well placed in brokering between stakeholder parties by showing the way forward through experimenting with the data and visually communicating experimental outcomes.

Defining primary keys

Primary keys serve as the link between the data records and their counterpart in the real world. Unfortunately, some business processes do not rigorously define primary keys. This can lead to slippery definitions in the database. In other cases, the primary key is defined at the application layer but remains invisible at the data layer, leaving the business user without a clear way to interpret the data.

The data engineer must always establish the primary key where it is not *meaningfully defined*. The primary key articulates the engineer's view of how the data row is to be interpreted as a business entity.

Most primary keys can be traced by examining how a business process creates, retrieves, and updates a data record. There are three common patterns worth noting: sequence numbers, version numbers, and temporality.

Sequence numbers

Sequence numbers are common when a header has multiple detail rows, and the details table lacks a meaningful primary key. For example, a sales order may have multiple items. The order is stored in a table called *Sales*, with [Order number] as the primary key. The items are stored in *SaleItems*, which may be a miscellaneous list without a key.

If each product can only appear once, then [Order number], [Product ID] may suffice as a primary key. But if a product can appear more than once, this fails. A simple solution is to treat the list of items as a sequence and create an artificial column [Sales item sequence number], forming the key [Order number], [Sales item sequence number].

Sometimes a system will implement a meaningless key like [Sale items ID], a simple integer used for UI or database constraints. Even in these cases, the data engineer

should define a sequence number and use [Order number], [Sales item sequence number] as the primary key. This is more expressive of business intent, while [Sale items ID] can be retained for joins.

In general, sequence numbers are effective wherever there is a miscellaneous list of line items within a header. Care must be taken to ensure the sequence is deterministic by breaking ties—using a surrogate key like [Sale items ID] as a sort order is a reliable approach.

Version numbers

Immutable entities are those that do not change, or if they do, the change is considered sufficient to treat them as a new entity. For example, an import declaration is immutable. If the importer changes the content ahead of arrival, it is treated as a new declaration.

Version numbers are an effective way to manage changes in immutable entities. For example, *Import.Declaration* may use [Declaration ID] as the primary key. If changes occur, they can be stored in *Import.DeclarationVersion* with the key [Declaration ID], [Declaration version number].

Digital systems are not always consistent in how they manage versions. A common but problematic approach is to allow the entity ID to change indefinitely and record overrides using [Override declaration ID]. This may work for rendering a web UI but causes confusion in analysis. In every case where versioning is lost or muddled, the data engineer must restore clarity using the consistent pattern [Entity ID], [Version number].

Temporality

Temporality refers to tracking changes over time. In data warehousing, this is known as a type II table. Some business processes are designed to handle only the current event, without tracking history. But when the business is interested in change over time, the entity is mutable.

Mutable entities should be expressed through a primary key that includes a time component. The data engineer should define the key as [Entity ID], [Start datetime], and include [End datetime] to mark the end of the validity period. This is especially important if an entity can be deleted and recreated and thus the start of one row is not necessarily the end of the previous.

Recovering temporality depends on how history is stored. Sometimes it is available in audit tables. Sometimes it must be reconstructed with help from business experts who can supply missing information.

Defining the primary record

A related but distinct phenomenon is that of defining the primary record. This occurs when the database does have a primary key, but different primary keys relate to the same underlying entity. It is common in business processes that gather information from decentralised observations.

Consider the example of whale sightings. A database may collect observations from citizen scientists, stored in a table called *Whale.Observation* with primary key [User ID], [Observation number] to identify each user's submissions. A surrogate key such as [Observation ID] may also exist in the digital system for record retrievals.

In this scenario, multiple citizens may report sightings of the same whale. As a result, multiple [Observation ID] values relate to the same real-world entity. Suppose the business has a rule such as "a whale species at one GPS proximity would only appear once within a day." This rule can be used to group observations and identify a representative record.

The data engineer can express this by identifying the primary observation. For example, selecting the first observation of the day at a location and storing it as [Primary observation ID] in a table called *Whale.PrimaryObservationId*. This table would contain two columns: [Observation ID] and [Primary observation ID], where [Observation ID] is the key for the original record, and [Primary observation ID] identifies the chosen representative. Subsequent transformation in the data pipeline can focus working on the [Primary observation ID] grain. It can also be used in Power BI to return the true count of whale sightings.

Identifying the primary record in cases of multiple observations is a powerful tool for helping the business interpret and relate to the data.

Care must be taken to resolve race conditions. If two users submit observations simultaneously, only one record should be selected as primary. This can be resolved deterministically using a surrogate key such as [Observation ID].

Defining relationships

Digital systems are usually good at recording relationships between data records as they are captured by business processes. However, business processes often record only the relationships needed to execute a workflow, and not the relationships that are useful for analysis. This is especially common when the system is designed for operational efficiency rather than analysis. The following looks at two useful techniques – nearest temporal joins and mapping tables.

Nearest temporal join

Sometimes two sets of events are related, but the relationship is not recorded as database keys. This is common when the business process captures only what is needed for operational execution, not what is useful for analysis.

If business knowledge suggests that one set of events is expected to precede another, the relationship can be recovered by identifying the nearest preceding event. This technique is known as a nearest temporal join.

Consider the case where a lounge allows members to check in, and the members make different purchases at the bar in the lounge, or none. The check-in is recorded by the door scanner with the check-in datetime. When the member checks in, the membership level – bronze, silver, gold, and diamond - is recorded at the door. This is stored in *Club.Checkin* with columns [Member ID], [Check-in datetime] and [Membership level]. There is a surrogate key [Check-in event SK].

Purchases at the bar are recorded separately at the register. These are stored in *Club.Purchase* with columns [Member ID], [Purchase datetime], and [Purchase item]. There is a surrogate key [Purchase event SK]

Although both tables contain [Member ID], there is no direct relationship between a specific check-in and the purchases made during that visit. The system does not record which check-in a purchase belongs to. This makes it difficult to identify the membership level associated with the purchase.

To support analysis, the data engineer can create a link using the **nearest temporal join**. For example, a rule might be: “a purchase belongs to the most recent check-in by the same member that occurred before the purchase.” The result would be stored in *Club.PurchaseCheckin* as a two column table of [Purchase event SK] and [Check-in event SK].

One implementation would be to identify the latest check-in for each purchase using a MAX() aggregation, then joins back to retrieve the membership level.

```
; with NearestKey as (
    select
          pu.[Purchase event SK]
        , max(ci.[Check-in datetime]) as [Latest check-in datetime]
    from        Club.Purchase pu
    left join   Club.Checkin ci on   ci.[Member ID] = pu.[Member ID]
                                 and ci.[Check-in datetime] <= pu.[Purchase datetime]
    group by pu.[Purchase event SK]
)
select
      pu.[Purchase event SK]
    , ci.[Check-in event SK]
from        Club.Purchase pu
left join   NearestKey nk on nk.[Purchase event SK] = pu.[Purchase event SK]
left join   Club.Checkin ci on ci.[Member ID] = pu.[Member ID]
```

This table re-establishes the relationship between the purchase event and the check-in event.

Another approach is to consider each check-in as a window from one to the next, and for each purchase, find the corresponding check-in window.

```
; with CheckinTypeII as (
    select
          ci.[Check-in event SK]
        , ci.[Member ID]
        , ci.[Check-in datetime]
        , lead(ci.[Check-in datetime],1,9999) over (partition by ci.[Member ID] order by
ci.[Check-in datetime]) as [Next check-in datetime]
    from        Club.Checkin ci
)
select
      pu.[Purchase event SK]
    , ci.[Check-in event SK]
from        Club.Purchase pu
left join   CheckinTypeII ci on ci.[Member ID] = pu.[Member ID]
                            and ci.[Check-in datetime] <= pu.[Purchase datetime]
                            and pu.[Purchase datetime] < ci.[Next check-in datetime]
```

Nearest temporal joins are especially valuable at the macro level, where business processes operate in bulk and lack unit-level linkage. For example, seed suppliers may distribute large batches to farms, and harvest yields are later reported in aggregate. While individual bags of seed are not tracked, it's still important to understand how seed characteristics such as supplier or variety affect outcomes. In such cases, the data engineer can define keys for seed batches and yield events, then use nearest temporal joins to associate each yield with the most recent relevant seed delivery.

Similar patterns arise in other domains. In regulatory compliance, businesses may submit periodic declarations or claims, while auditors conduct inspections or reviews at later dates. These audits are not tied to specific submissions but are temporally related. Nearest temporal joins allow business to associate each audit with the most recent relevant declaration, enabling analysis of compliance patterns over time.

Mapping tables

Mapping tables are introduced when the original system does not record relationships needed for analysis. A common scenario is a header table with two separate detail tables at different grains. Both detail tables relate to the header, but not to each other. The data engineer may introduce a mapping table between them based on a business rule.

Consider a club restaurant with data captured as follows:

- *Club.TableSitting* — records of customers seated at the restaurant, one per sitting, keyed by [Table sitting ID]
- *Club.TableCustomer* — club members identified by [Member ID], linked to [Table sitting ID]
- *Club.TableFoodOrder* — menu items ordered per [Table sitting ID], with food items identified by [Food item ID]. Items may be ordered multiple times

The system does not record which member ordered which item. For analysis, the business may wish to associate members with food items. However, this relationship is not captured in the source system.

To support analysis, the data engineer may introduce a mapping table *Club.CustomerFoodOrderMap*, with three columns [Table sitting ID], [Member ID] and [Food item ID]. The logic may be based on a business rule such as “every member at the table shares the cost of all items ordered.”

For implementation, it suffices to join *Club.TableCustomer* and *Club.TableFoodOrder* on [Table sitting ID]. However, because a food item can be ordered multiple times, the composite columns [Table sitting ID], [Member ID], and [Food item ID] do not yet form a primary key. Instead, additional group by on those columns, and additional window calculation is necessary for [Food item occurrences], which is the number of times the food item was ordered at the table, and for [Number of members], which is the number of members seated at the table. The ratio of these two values can be used as distribution weights in subsequent calculations.

An example implementation in SQL is as follows:

```
with FoodItemCount as (
    select
        [Table sitting ID]
      , [Food item ID]
      , count(*)            [Food item occurences]
    from Club.TableFoodOrder
    group by [Table sitting ID]
           , [Food item ID]
)
select
    tc.[Table sitting ID]
  , tc.[Member ID]
  , fc.[Food item ID]
  , fc.[Food item occurences]
  , count(distinct tc.[Member ID])
      over
    (partition by tc.[Table sitting ID])        [Number of members]
from        Club.TableCustomer   tc
inner join  FoodItemCount        fc on fc.[Table sitting ID] = tc.[Table sitting ID]
```

As an example of usage, the total cost spent by all members on each food item over time can be returned through:

```
select
    fm.[Member ID]
  , fm.[Food item ID]
  , sum(
      rm.[Food item cost] *
      cast([Food item occurences]/[Number of members] as float)   --distribution weight
    )                                                [Total food item cost]
from      Club.CustomerFoodOrderMap fm
left join Club.RefMenu              rm on rm.[Food item ID] = fm.[Food item ID]
group by  fm.[Member ID]
        , fm.[Food item ID]
```

Because the relationship is inferred rather than recorded, it should be tested. Techniques for validating mapping logic are described in the chapter *Tests and assumptions*.

The suffix *Map* can be casually used so that it becomes meaningless. It is more effective if it is reserved for genuine many-to-many relationships. If the relationship is one-to-one or expresses a property, the table should be named accordingly. For example, in the case of nearest temporal join, the table is *Club.PurchaseCheckin* because each purchase has only one check-in, and thus the word *Map* is not necessary.

Fuzzy logic

There are cases where precise rules do not apply. A common example is natural language processing. When working with free text, it is often impossible to define exact rules that meet business intent.

Fuzzy logic is increasingly common and accepted. The rise of large language models is accelerating this trend. In many scenarios, “something is better than nothing” as far as business objectives are concerned, and a perfect match with reality does not justify the cost. This aligns with the view that data quality issues are not about data versus reality, but data versus business intent.

Examples of fuzzy logic include optical character recognition to extract information from images, large language models to infer sentiment from user text, and entity resolution techniques to identify the underlying entity behind multiple records, such as in guest check-out scenarios.

Fuzzy logic pushes into the domain of data science, where statistical and mathematical disciplines bring rigour to its application. However, there are cases where a data engineer can apply a simple approach. The following is a three-step process — a kind of “poor man’s data science”:

- Find the pattern through **loose–tight iteration**
- Check the pattern with **random validation**
- Monitor the pattern by **watching for drift**

The approach can be highly effective in many common scenarios that requires extracting information from free text. A data engineer should be comfortable to consider this three-step approach and be ready to stay within it or pivot to a more sophisticated data science method when necessary.

Loose-tight iteration

A useful way to think about fuzzy logic is through the idea of **loose–tight iteration**. Every pattern has matches and rejects. The matches are records that meet the criteria. The rejects are those that do not. The data engineer tunes the pattern by adjusting the criteria—tightening to reduce over-matching, loosening to recover missed records. The goal is not to maximise matches, but to eliminate mistakes on both sides. This is an approach that avoids blind spots.

Consider the example of extracting phone numbers from a free-text field. The phone numbers may come in a format of area codes and standard spacing, such as (02) 1234 5678, but also other variations such as 0412345678, 12345678, or 02-1234-5678.

The goal is to extract the phone numbers from the free-text field. The challenge is that the field may have other numbers such as invoice numbers or date numbers.

Finding the best pattern is a matter of iteration of starting with a rough pattern, and refining. The following is a procedure template for arriving at a pattern.

1. Define a rough pattern
Start with a simple pattern that captures the most obvious cases.
Example: Match phone numbers in the format (XX) XXXX XXXX.

This pattern will produce:

- A **match set** — records that meet the pattern
- A **reject set** — records that do not

2. Inspect the match set for false matches
Review the matched records. Are there entries that should not be matched?
Example: The pattern may pick up numeric strings like 12345678 that are not phone numbers.

If so, tighten the pattern to exclude these.

3. Inspect the reject set for false rejects
Review the rejected records. Are there valid entries that were missed?
Example: Mobile numbers like 0412345678 or formats like 02-1234-5678 may be valid but were excluded.

If so, loosen the pattern to include these.

4. Adjust and repeat
Each adjustment changes the match and reject set, being careful to tighten or loosen at one time, but not both. Repeat the inspection after each change.
Continue until the pattern behaves acceptably — where the matches are genuinely useful and the rejects are genuinely not.

The key idea is that focusing solely on matches would lead to bias or blind spot. Instead, the best pattern achieves an optimised balance by looking at the incorrect matches and incorrect rejects. This approach is related to the data science discipline of optimising the F1 score.

Random validation
Once the pattern has been tuned, it should be validated by a business expert. This is done by randomly sampling records from both the match set and the reject set. The sample must be unbiased. It should not focus only on records where the engineer has low confidence, nor only on records that seem easy to validate. The point is to test the pattern across its full range of behaviour.

This process should be repeated periodically. Even if the pattern was correct at the time of implementation, changes in business processes or user behaviour may cause it to degrade. Regular validation helps ensure the pattern continues to serve its intended purpose.

Monitoring for drift

In practice, random validation cannot be performed continuously. Drift monitoring provides a lightweight alternative. The idea is to identify a statistic that should remain relatively stable over time. If the statistic changes significantly, it may indicate that the pattern is no longer behaving as expected.

For example, suppose a free-text field is used to extract phone numbers. Not all extracted numbers will match the customer database, but a certain percentage—say 85%—typically do. This percentage reflects the stability of the pattern. If it drops to 60%, it may indicate that users have started entering phone numbers in a new format, or that the pattern is picking up irrelevant content.

This is a form of assumption monitoring. The assumption is that the extracted values will continue to resemble the known population. If the assumption fails, the pattern may need to be revisited. Multiple statistics can be used to provide greater assurance.

Drift monitoring is especially useful when the business has a relatively static reference point such as a customer table, a list of known codes, or a set of standard formats. These provide a benchmark for assessing whether the pattern is still aligned with business intent. However, any stable statistic can be used.

Tests and assumptions

The world is constantly changing. What is true today may not be true tomorrow. New data may arrive that fall outside previously conceived parameters. Business processes may evolve to meet emerging needs. Or another developer may alter existing code. For these reasons, the data engineer must continuously monitor for disruptive change.

The world is also complex. Mistakes are easy to make. A data engineer may misinterpret business logic, overlook edge cases, or introduce technical errors when writing complex code.

Thus, the third principle of data engineering is **anticipating errors**. Rather than focusing solely on what is working now, the data engineer must remain attuned to what may go wrong. Two explicit ways of doing this are to design thoughtful tests and to monitor assumptions.

Tests and assumptions need to run regularly, such as once per data pipeline batch run. See the chapter on *Automation*.

Thoughtful tests

The premise of testing is this - when the same problem is solved in two completely different ways, in a completely independent manner, by two different people who arrive at the same result, confidence in accuracy is justified. In other words, **tests mean doing something twice, in two different ways.**

Consequently, a test has two parts. The first part is a query that returns the expected results. The second part is a query that returns the actual results. The test pass if they match exactly.

In a data pipeline, the expected part of a test may be a query calculated from pre-transformed data, either raw or partway through the pipeline. The actual part uses post-transformed data. A simple example is comparing row counts between raw and curated data.

In a Power BI semantic model, the expected part of a test calculates results from the underlying data sources, such as SQL or a data lake, while the actual part calculates the same results using DAX. This approach is highly effective for validating complex DAX measures, especially those referencing hidden grain (see *Designing Measures*).

Power BI tests are easy to create. The data engineer can drag and drop to build a table visual with dimensions and measures. The Performance Analyzer in Power BI Desktop reveals the DAX query behind the visual, which serves as the actual portion of the test. The engineer then matches this DAX evaluation with a calculation from the underlying data source.

s 22(1)(a)(ii)

Tests lose effectiveness when the expected query is a copy of the pipeline code that produced the actual results. This entanglement reduces independence and narrows the surface of possible errors caught by the test. A thoughtful test maximises the difference in logic between expected and actual results. For example, the expected part for a Power BI test should use fragments from the pipeline rather than the tables or views that directly feed Power BI. This helps catch errors in data source definitions.

The way to write a good test is to ask, “Where is the most complex or most fragile part of the transformation, and how can this transformation, or a part of this transformation, be calculated in a completely different way?”

The following are some ways for introducing independence in calculation to write thoughtful tests. They are not intended to be exhaustive, but intended to illustrate the mindset of checking weak points of an implementation by performing the same calculation completely differently to the original implementation.

Row counts

The simplest test is to compare row counts before and after transformation. The expected query counts rows from the raw data with appropriate filters. The actual query counts rows from the transformed output.

Even small variations can make this test more effective. One approach is to group by a reference value and vary how the reference is looked up. For example, the expected query might use an inner join, while the actual uses a left join. If both queries normally return the same result, any discrepancy indicates an error.

Row count tests are especially important for incremental loads. The expected and actual queries should count recent rows, but the selection must use a datetime column different from the one used for extraction. For example, if the extract relies on an architectural column like [Row update datetime], the test should use a business-centric column such as [Staff update datetime]. This separation avoids entanglement and self-confirmation. These row counts act as checksums for create and delete operations. Grouping by reference values helps catch update errors on those columns. Selecting a sample of recent records is a valid alternative that would check CRUD on every value.

Checksum on key results

Consider the case where each entity is either “bad” or not. For example, a manufacture batch is considered bad if it fails one or more quality control criteria. The data engineer should summarise these results back to the batch level for use in a storytelling dimension.

Suppose the original tables are:

- *Cake.Manufacture* — details of manufacture batches, at [Manufacture batch number] grain
- *Cake.QualityControlResult* — quality control results per batch, with [Is fail] indicating failure

These are summarised to:

- *Cake.ManufactureQuality* — aggregates results to [Manufacture batch number], where any failure marks the batch as failed
- *Cake.RefManufactureQuality* — reference table with [Has any failure in batch] as the summary flag

To test the aggregation logic, the test can use a checksum. Note that a batch is either bad or not. So, the total number of batches minus the number of batches with at least one failure should equal the number of batches with no failures.

In SQL terms:

Expected:

```
SELECT COUNT(*) FROM Cake.Manufacture
- SELECT COUNT(DISTINCT [Manufacture batch number]) FROM Cake.QualityControlResult WHERE [Is fail] = 1
```

Actual:

```
SELECT COUNT(*) FROM Cake.ManufactureQuality WHERE Cake.RefManufactureQuality.[Has any failure in batch] = 0
```

This test checks the core logic without repeating the original transformation. It confirms that all batches are accounted for and that the aggregation behaves as expected. If a future developer modifies *Cake.ManufactureQuality*, for example to optimise performance, any accidental errors would be caught immediately.

In more complex scenarios, this test also helps surface other weaknesses. For a more comprehensive check, the query should group by values such as [Product type].

Bypassing mapping tables

Mapping tables are often introduced to implement business logic where the original system did not record relationships. A good test for the mapping table is to compare checksums with and without it.

Recall the example from *Dealing with data quality*. A club restaurant with data captured as follows:

- *Club.TableSitting* — records of customers seated at the restaurant, one per sitting, keyed by [Table sitting ID]
- *Club.TableCustomer* — club members identified by [Member ID], linked to [Table sitting ID]
- *Club.TableFoodOrder* — menu items ordered per [Table sitting ID], with food items identified by [Food item ID]. Items may be ordered multiple times

The system does not record which member ordered which item. For analytics, a mapping table such as *Cake.CustomerFoodOrderMap* may be introduced to associate [Member ID] with [Food item ID] per [Table sitting ID].

Regardless of how complex the mapping logic is, the total number of member attendances, food items ordered, and the sum of food item cost must remain unchanged. An effective test compares these totals before and after applying the mapping. For example, the expected query sums these values directly from the source tables. The actual query performs the same aggregations using the mapping table.

Grouping by reference columns such as [Food item type] can enhance the test further.

This is a powerful way to validate mapping logic. Any accidental errors—especially in complex mappings—are immediately caught by such a checksum. A more detailed test can be tailored to check the specific logic itself. For example, if the business rule assumes the food cost is evenly distributed amongst the members, then the test can check distribution weight is the same for each member.

Checking boundary cases

Boundary cases are common sources of error. A typical example is a rolling window measure. Any implementation of rolling windows should be tested at boundary points.

Suppose there is a Power BI measure [Total sales volume], and a derived rolling 12-month version [Total sales volume past 12 months]. Errors in the latter are easy to miss.

One observation is that both measures should coincide at specific boundaries, such as the end of each financial year and calendar year. An expected query can group [Total sales volume] by calendar year, while the actual query groups [Total sales volume past 12 months] by the same. The results should match, even at the beginning of the history where fewer than 12 months of data exist.

However, this test would still pass if [Total sales volume past 12 months] were mistakenly implemented as a copy of [Total sales volume]. To guard against this, additional checks should be performed at turning points—such as the first and last day of a month. For example, selecting 28 February and 1 March helps test the offset around calendar boundaries.

A rolling window measure that passes at the end of each calendar year, the end of each financial year, and the first and last day of an interim month provides strong assurance. These checks can be calculated independently of how [Total sales volume past 12 months] is implemented.

Independence of subject matter knowledge

Where subject matter knowledge is available, it can introduce independence into a test. Normally, the data engineer should avoid assumptions when building the pipeline. This prevents blind spots and reduces the risk of silent errors. However, unit tests are one place where business knowledge can be safely used to strengthen robustness.

For example, a business expert may know that every header must have at least one detail. In this case, a left join and an inner join between the header and detail tables would return the same result. The data engineer should not assume this in the pipeline, since rare load errors may cause headers or details to be missing. But the knowledge can be used in a test to validate the relationship.

Business knowledge can also be more complex. Consider a help desk system where cases escalate from Tier 1 to Tier 4. Each escalation is recorded in *Helpdesk.Escalation*, keyed by [Case ID], with [Tier] indicating the escalation level. The table *Helpdesk.CaseEscalation* rolls up this information to [Case ID] grain. A reference table *Helpdesk.RefCaseEscalation* includes [Highest escalation] to summarise each case.

Tests can use business knowledge to validate the roll-up logic. For example:

Business knowledge	**Expected**	**Actual**
The maximum tier is “Tier 4”	Distinct count of *Helpdesk.Escalation*[Case ID] where [Tier] = “Tier 4”	Count rows of *Helpdesk.CaseEscalation* where *Helpdesk.RefCaseEscalation*[Highest escalation] = “Tier 4”
Every case starts at “Tier 1”	Distinct count of *Helpdesk.Escalation*[Case ID] where [Tier] = “Tier 1”	Count rows of *Helpdesk.CaseEscalation*

Cases *do not* downgrade from any tier	Count rows of *Helpdesk.Escalation* where 1) [Tier] = “Tier 1” and 2) [Tier] = “Tier 4” respectively	Count rows of *Helpdesk.CaseEscalation* where 1) there are no filters and 2) [Highest escalation] = “Tier 4” respectively

In general, existence and uniqueness conditions known to the business expert can be used to bypass or vary the original pipeline logic. These checks help catch implementation errors or faults in load mechanics. However, the business expert may also be wrong—especially when edge cases are forgotten. By introducing business knowledge into a test, the data engineer can validate the claim against the data. If the knowledge was correct at the time of implementation but later becomes invalid, the test will help detect the change in business process. Consequently, whenever business knowledge is available, they should be injected into tests.

Monitored assumptions

Data engineers must make assumptions. However, these assumptions may fail over time. Sometimes the failure is not anticipated. At other times, the failure is anticipated but must still be detected. In general, users require automated alerts for events that need attention. These are managed through **monitoring assumptions**.

An assumption has one part. It is a query that returns a non-empty result set if and only if human attention is required. Each row that returns from the query is a violation of the assumption of a business logic that requires action.

The key to monitoring assumptions is recognising that an assumption exists. The following are common cases to help data engineers get started.

Is source data up to date?

Source data cannot always be assumed to load on time each batch. Any number of failures may occur to delay the arrival of new data. An assumption can check that the latest data has arrived and alert the engineer or the user that the latest data has not arrived as expected.

Are reference data complete?

Reference data is vital to a quality warehouse. It enables diverse systems to map to a conformed set of golden records which enables integration across diverse business. These mappings require constant maintenance as new data comes in that requires mapping. An assumption checks daily for new records that require mapping.

Are there unanticipated data?

Hard-coded values are sometimes unavoidable. This is acceptable if the source column contains, for example, six values that have not changed in a decade. A transformation may map these values to categories. If a new value appears, it likely signals a significant change in transactional logic. An assumption should check for new values.

Are there data quality issues?

Source data may contain issues from unvalidated collection. These can include duplicate keys or values outside normal ranges. For example, most years may be entered correctly, but on occasions, 2025 may be recorded as 2205 by accident. During transformation, discarding or altering these records may be acceptable. An assumption should check for such anomalies so they can be raised with the business owner for correction at the source.

Is there data drift?

Mathematical analysis relies on assumptions about the statistical nature of input data. These assumptions can change over time—a phenomenon known as data drift in machine learning. Data engineers may encounter this when applying fuzzy pattern to extract information from free-text fields. Such logic often assumes a word pattern to be used. Instead of a “set-and-forget” approach, data engineers must monitor for drift in pattern and intervene when necessary.

Conclusion

A test performs the same calculation in two different ways and checks whether they return the same result. If the data engineer avoids the entanglement effect, this vastly reduces the likelihood of undetected errors—whether they stem from changes in the world or mistakes in the engineer’s logic.

An assumption is a calculation that returns a list of violations to business logic that were presumed to hold. If no rows are returned, the assumption remains valid.

Any condition can be expressed either as a test or an assumption by rephrasing. The rule of thumb is to express critical conditions as tests, and slow-moving or less critical conditions as assumptions. For example, row counts before and after a transformation should be expressed as tests — a discrepancy may indicate a serious error. New country codes requiring annotation can be expressed as assumptions — a failure may affect only a small number of newly arrived records.

Modern development promotes continuous delivery to produce quality, fit-for-purpose products for stakeholders. The rhythm is fast-paced, with developers under pressure to deliver weekly or even daily. In this context, writing tests and assumptions may seem to

slow down delivery. In reality, the opposite is true. Rapid changes to complex code increase the likelihood of significant errors. When errors occur, they are costly to fix and can damage the team's reputation as a provider of trusted information.

Consequently, while writing tests and assumptions takes development time, it ultimately accelerates delivery by mitigating the risk of costly mistakes and enabling the team to deploy changes with confidence. While thoughtful tests are best, even simple tests are valuable. In practice, most errors are careless mistakes made during rapid development that simple tests can catch before deployment.

New engineers naturally spend more time choosing patterns and building their implementation. With experience, design becomes mechanical and rapid. As a result, mature engineers spend proportionately less time responding to current needs and more time anticipating errors. A rough guide is to allocate about 25% of each development cycle to defining tests and assumptions.

Fault tolerance

Tests and assumptions can be understood as an aspect of fault tolerance, which itself is an aspect of error handling.

Fault tolerance is the idea that an error in one part of the pipeline should not cause the whole to fall over. This means that errors in a few rows should not prevent a table from loading. It also means that failure in a few tables should not halt the entire pipeline. In a large-scale pipeline with hundreds or thousands of tables, the probability of errors occurring in each batch is high. Consequently, fault tolerance becomes a necessity, not an optional extra.

The goal of fault tolerance is not to eliminate all errors. The goal is to ensure that the pipeline continues to function in the presence of errors, and that errors are surfaced in a way that is meaningful and actionable.

This chapter introduces three common fault patterns:

- Uniqueness
- Existence
- Stability

Each pattern is accompanied by practical techniques for detection and mitigation. While broad, these patterns are not exhaustive. Rather, they illustrate a mindset of building a fault tolerant pipeline.

Uniqueness

If the basic task of data engineering is to relate real-world objects and processes into the data world, then the most common quality problem is that of uniqueness and existence. Uniqueness dictates that one record in the real world corresponds to at most one database record, and vice versa.

Uniqueness is violated when one real-world entity appears as two records in the database. This can happen when a staff member accidentally enters the same transaction twice.

The reverse is also true. Uniqueness is violated when two distinct real-world entities are forced to occupy the same database record. This may happen if a staff are identified by first and last name. If a staff member joins who uses an existing name, the two staff will have the same identifier value in the database.

Uniqueness violations can come from failures in business processes, as in the examples above. It can also come from mechanical failures, such as an incremental load logic incorrectly loading the same records twice

The data engineer must express all uniqueness expectations in the warehouse. These expectations are not just technical constraints. They are best seen as statements of business intent.

The simplest way to express uniqueness is through a unique constraint in the database. If the constraint is violated, the pipeline has two options:

- Abort the load and leave the data as it was; or
- Apply automatic de-duplication, send the duplicates to a reject table, and allow the rest to proceed.

The second approach is explained in the chapter *Load mechanics*.

In either case, the violation must be surfaced. A monitoring report should alert the user. If the violation is frequent, a different treatment is necessary. Repeated alerts can lead to the “cry-wolf” effect of dulling attention. In such cases, the data engineer should implement targeted handling for the table. For example, instead of relying on a uniqueness constraint, the pipeline can perform de-duplication during transformation and store duplicates in a separate table for review. Such techniques are illustrated in *Dealing with data quality*.

Expressing uniqueness expectations consistently is one of the most effective ways to improve the correspondence between the data world and the real world. It is also one of the most direct ways to build trust.

Existence

The criteria for existence can be handled similarly as uniqueness. Existence dictates that the record must exist in the database if the object exists in the real world, and vice versa. Sometimes this “object” can be a property or an attribute. For example, a completed sales record must have a sales amount.

Existence is violated when the real-world entity exists, but it does not occur in the database, as in the case if the staff has forgotten to record the sales amount. It is also violated when a database record exists, but the real-world entity does not, as in the case when a record fails to be deleted during an incremental load.

While uniqueness violation tends to come from failure in the business process, existence violation often comes from the mechanics of data warehousing. For example, during batch loads, the *Cake.SaleItems* table may be extracted at a different time to the *Cake.Sales* table, leading to inconsistency where one exists without the other. In traditional warehousing this is called late arriving transactions.

Existence can be expressed in the database through the "not null" constraint on a column. In the case of sales requiring sales amount, it is a matter of applying the not

null constraint on the *Cake*.*Sales*[Sales amount] column. This is not just a technical safeguard. It is a statement of business intent.

In more complex cases such as the *Cake*.*SaleItems* detail table requiring the existence of rows in the *Cake*.*Sales* header table, then the detail can left-join to the header, and apply a not null constraint on the header ID. This would alert the engineer if the detail ever missed a header.

When encountering a violation, the data pipeline can apply similar automation as that of uniqueness by sending violating rows into a reject table and allowing the rest to proceed.

As with the case of violation, repeated occurrences of the existence violation will need more targeted handling. For example, instead of left joining *Cake*.*Sales* and *Cake*.*SaleItems*, an inner join would ensure that no sale items are presented if they are missing a header sale record. However, this treatment should not lead to a silencing of error. If the records do not catch up, suitable tests should alert of prolonged discrepancies. In the case of incremental load, extra attention is needed to ensure records catch up.

For the case of existence, the best mindset for an engineer is to build the pipeline as if it were streaming. In this case:

- Tables may load continuously, or at least a few times a day
- Some may not load from time to time
- Tables may load in random, out of order

In these cases, existence of records may be violated in unpredictable ways, but the pipeline should handle these cases by catching up in the next batch. This is the mindset that would lead to the most fault tolerant pipeline.

Stability

Stability is a concept from mathematics and from physics. In computational mathematics, it refers to the idea that small changes in inputs should lead to small changes in outputs. In physics, it's the difference between a ball on top of a hill versus a ball at the bottom of the valley. A ball on the top of the hill is instable while the one at the bottom is stable. This is because small perturbation in the former will lead to the ball rolling off, while perturbation in the latter will lead to the ball returning to its original position.

In similar fashion, a data pipeline needs to be stable. This means that small changes in the real world should lead to small changes in the database world. Particularly, small changes in information should lead to corresponding level of create, update, delete (CRUD) operations.

However, while the real world rarely has massive number of changes — that is, users rarely go back and rewrite all the historical records — the database world often suffers from massive changes that are accidental rather than real. This can happen during server updates that cause the [Row update datetime] values to all change.

If uncontrolled, accidental changes that lead to large changes can significantly harm stakeholder confidence. For example, failing to load the country reference table, which is a small table of 100 or 200 records, leading to complete disappearance of country data in the warehouse.

In addition, such uncontrolled operations can also bring down a server when there are too many CRUD operations than normal.

Consequently, a developer has the responsibility of ensuring that the pipeline is built so that its changes are commensurate with real world changes. Sudden increase in changes may indicate an error.

To build such a pipeline, the data engineer needs to avoid instable elements. For example, a column called [Today's date], which is sometimes used for reporting, on a table will cause every row to change every day. Columns that are non-deterministic should also be avoided. Non-determinism often creeps in unintentionally when a column's value is determined by a ranking, and the ranking columns are not sufficiently specified to return the same value every run. This leads to rows changing despite no change in the real world. Moreover, single row tables such as *Cake.LatestLoadDate* that contain the latest load of the data should be avoided. The same effect can be achieved with appending data rather than overwriting.

Violations of stability can be caught by stability thresholds. For example, the data pipeline should abort the table's load if more than 5% of the records are to be deleted. The implementation of this is explained in *Load mechanics*.

Apart from the automation with stability thresholds, data engineer habits are also important. One habit is to avoid having wide tables with miscellaneous attributes. In such a table, change in any cell value causes the whole row to update. This effectively amplifies the impact of real-world changes to database changes. Instead, data engineers should build table fragments so that each table has clear meaning, and so changes to that table's rows correspond to those meanings.

Another habit is to avoid placing reference tables at the beginning of the data pipeline. Accidental small changes to reference tables would have massive impacts downstream, as is the case of the country codes.

More of these concepts in the *Efficient & stable pipeline*, particularly in *Load mechanics* and *Dependencies*.

Conclusion

Uniqueness, existence, and stability violations are three common patterns of faults. They can be handled in a generic way using standard automation, but also through specific pipeline design in the case of frequent violations. Good habits also improve fault tolerance. With fragmentation, tests, assumptions, narrow tables, and thoughtful dependency management, the data engineer has an arsenal of tools to handle common problems.

However, the possibilities of errors are limitless. For example, in parallel loads or a busy server, we may never be able to fully eliminate deadlocks. And thus we will need to implement deadlock retries.

It will not be fruitful to enumerate all the errors. Instead, the focus is on the mindset, rather than on the set of techniques. The experienced data engineer does not focus solely on what is working right now, but also thinks about what can go wrong. This is the third of the four data engineer principles — anticipate errors.

The final comment on fault tolerance is that the engineer must have a mindset of expecting errors, learning from errors, and writing code that can capture them. Developing this mindset will take time. The best guide for the engineer is well stated by the Zen of Python: **“Errors should never pass silently. Unless explicitly silenced.”**

Efficient & stable pipeline

Efficiency and stability

Developers often talk about fast and slow. But what do these terms mean? Is one second fast? Is ten seconds too slow? There are situations where one second to process ten rows is unacceptably slow, and others where ten seconds to process ten thousand different rows is impressively fast. These judgements depend on the infrastructure, the structure of the incoming data, and the value of the information being processed.

A sharp intuition for whether something "can still be faster" is one of the hallmarks of a veteran developer. A developer who cannot tell that something could be faster will not attempt to improve it. On the other hand, trying to optimise a computation that is already at its limit is futile.

Given the number of environmental factors involved in time-to-compute, and how slippery the idea of "fast" can be, this intuition is difficult to develop. A way to make the problem more tractable is to focus on one aspect of performance — efficiency.

Efficiency in a data pipeline means using the minimal amount of resource to calculate one piece of information. There are two ways to achieve this:

1. The pipeline processes the minimal amount of information needed to reflect changes in the data.
2. The algorithm that applies these changes uses the minimal amount of computational space-time resources.

The first is informational efficiency. The second is algorithmic efficiency.

Informational efficiency is about processing as little information as possible. A pipeline is inefficient if it processes hundreds of millions of rows but in the end, only a handful changed. Algorithmic efficiency is about computing those changes as cheaply as possible on a predefined computational environment. It is inefficient if it takes seconds to process one row when it could be done in milliseconds. Informational efficiency is universal across all technologies. Algorithmic efficiency is more technology dependent and varies from site to site.

Efficiency matters because it is key to the sustainable growth of a warehouse. It also directly affects how quickly users get their results. It is within the data engineer's control. Decisions about server configuration and information value may not be easy to influence. However, the data engineer can always ensure a pipeline does just enough work, and no more, to reflect real world changes.

Stability is a closely related concept. It is the idea that small changes to input should lead to small changes in output. An unstable pipeline is one where a small change leads to a large change in output. For example, changing a country name in a reference

table might ripple through and rewrite many tables. A more dramatic instance of this was the fear of the Y2K bug where there were widespread concerns that switch over on the New Year's Day 2000 would cause disastrous impacts on systems that stored years only in the final two digits.

Stability matters because it shapes the user's experience of reliability because of the magnitude of impact makes the problem obvious. Reports that suddenly become empty one day radically undermines trust. There is no point explaining it was because a change in international country names caused a ripple effect. As far as the user is concerned, the product is unreliable.

Efficiency and stability are related but distinct. An inefficient pipeline is when there is a lot of work, but few results change in the table. An unstable pipeline is when there are too many changes in a table. From a business point-of-view, instability can also be inefficiency. For example, a large table with [Today's date] column would be updating every day (unstable) but is inefficient because it is much work for little business value. That is, the number of row changes do not reflect information value changes.

The chapters in this part explore of efficiency and stability. A recurring theme is that meaningful fragments form the foundation of an efficient and stable pipeline.

Load mechanics

During a load, a table's data needs to update with the most recent set. The simplest approach is to drop the original table and replace the data with a new set. This is how every data engineer starts. But it violates the key tenets of pipeline efficiency and stability.

When a table's content is replaced en masse, it is not possible to know which rows changed. Downstream processes cannot respond with precision because every row looks new and needs response. This breaks the goal of efficiency to ensure database changes are minimal.

More importantly, when a table is dropped and replaced, any error that prevents a successful load becomes catastrophic. Reports may become empty for the user. It can also result in a domino effect of errors in downstream tables.

In a production pipeline that transforms hundreds or thousands of tables, the cumulative probability and effect of these problems become untenably disruptive.

Instead of drop and replace, updating the data of a table should occur through three steps:

1. **Stage** the incoming copy of the data into a temporary location
2. **Check** the incoming data against current data to pick up changes, anomalies and constraint violations
3. **Apply** genuine changes to the target table, updating any timestamp columns.

These are the same steps whether the incoming data is a full extract or a subset of increments.

Loading

Suppose the current table is *X*. The goal is to update *X* with the latest batch of data. *X* has a primary key column [PK col1], [PK col2].

Step 1 – Stage

The first step is to load the latest batch of data into a temporary table. In traditional warehousing terminology, this is the staging table. Suppose it is called *X_staging*.

X_staging may be the full load of data to replace *X*. It may be less than the full set but still cover all the changed records in a recent batch. This is an incremental extract. An incremental extract reduces the number of records to stage, check, and apply from hundreds of millions to a couple of thousand. The ways to do this reliably are covered in *Tracking change* and *Responding to change*.

Step 2 – Check

The next step is to check the content of *X_staging* against *X* for changes and validity before loading onto *X*. This is the most intensive step and is the one usually skipped in naive approaches to loading.

The aspects to check are:

- Changes:
  - Which rows to insert?
  - Which rows to update?
  - Which rows to delete?
- Instability:
  - Are there a concerningly high number of rows to insert or update?
  - Are there a concerningly high number of rows to delete?
- Violations:
  - Are there rows of uniqueness violations to reject?
  - Are there rows of existence (not null) violations to reject?

Checking for changes is essential. Checking for instability is highly desirable. Checking for each of the violations is not critical, but proactively rejecting a small number of bad rows and allowing correct rows to pass through can reduce disruption to users by allowing them to get access to most of their data.

Checking for changes

Not every row in *X_staging* is a genuine change in *X*. In fact, during a full load, only a few hundred out of millions may differ in the latest batch. The first step is to detect which rows are the ones that changed.

This check is necessary even if *X_staging* is incrementally extracted with the intention of capturing only changed data. Despite the best effort of the data engineer, it is often difficult to extract only the changed rows, especially when there are multiple joins and transformations. While incremental extract may narrow the number of candidate rows from millions to thousands, some of them may still be identical to the rows in *X,* requiring a check to find those that are truly changed.

In checking for changes, there are three possibilities:

- For a row in *X_staging*, is the primary key value new? If so, it is an **insert**.
- For a row in *X_staging* that has an existing primary key value in *X*, are any of the column values different? If so, it is an **update**.
- For a row in *X*, should the primary key value still be retained? If not, the row should be **deleted**.

The first two are collectively known as an **upsert**. A computationally effective way of calculating inserts and updates simultaneously in SQL is to:

1. Start from *X_staging* and left join *X* on the primary key columns [PK col1], [PK col2].
2. Keep only the rows from *X_staging* that do not identically exist in *X*. This can be done using an EXCEPT comparison in the WHERE clause.
3. Add a computed column [Is new row] depending on whether the *X* primary key column value is null.

Storing the result of this query into *X_upsert* gives only the rows that changed, with a column [Is new row] to indicate whether it is an insert or update.

[EXAMPLE SQL]

Checking for deletes depends on whether *X_staging* is fully extracted or incrementally extracted.

If *X_staging* is fully extracted, then it represents all the records that should be in the final table. Rows from *X* that should be deleted are those whose primary key values are not in *X_staging*. An implementation in SQL would be a left anti-join of *X* on *X_staging*. The primary keys to delete should be stored in a table *X_delete*.

[EXAMPLE SQL]

If *X_staging* is incrementally extracted, the idea would not work because it only has a fraction of the primary keys in the desired final output. Rather, *X_delete* needs to be tailored to the incremental extract logic for *X*. This is explored in greater depth in *Responding to change*.

Checking for instability

With the tables *X_upsert* and *X_delete*, it is easy to check whether there are an abnormally high number of rows to be upserted or deleted in a batch. If there are an abnormally high number of rows, then the load should abort.

The thresholds for abnormality can be a combination of the number of rows in the base table and the percentage of rows. These are stability thresholds.

For example, if the table *X* has a hundred thousand rows, and more than 10% are upserted, then this can be a sign of abnormality for aborting the load.

The threshold for deletes should be different from the threshold for upserts. There are usually far fewer deletes than upserts. For example, less than 5% of a large table should be deleted on a given day.

In practice, a simple stability threshold can apply as a default across the entire pipeline to catch most abnormalities, such as having half the rows disappear. Specific tables may have their stability thresholds tailored to their case. Examining a table's history of

changes can be useful in determining the threshold. This can be done if these statistics are logged. This logging is discussed in a subsequent subsection.

The SQL for aborting on the upsert threshold can be done by checking the number of rows in *X_upsert* versus that of *X*:

[EXAMPLE SQL]

Checking the delete threshold is only possible on a full extract. The SQL is as follows:

[EXAMPLE SQL]

These checks are an important part of monitoring and improving a pipeline's stability. It is very unlikely that every record of a business process's entire history, or even more than 5% of them, changes their content. If the upsert rows in *X_staging* do exceed this number, it is more likely to be an unintended side-effect of a system change rather than a change in informational content. Stability thresholds prevent these unintended changes from propagating.

If the load is frequently aborted for a table, then the extraction logic needs to be improved for stability, or the thresholds need to be adjusted.

Checking for violations

When there are too many rows to upsert or delete, the entire table aborts its load. This means the user does not get any of the latest data. This can be too drastic when only one row out of thousands is problematic.

Instead, if the table has uniqueness constraints and existence constraints (not null property on a column), only the rows which violate these constraints need to be discarded, and the rest can proceed. These are the reject rows and can be stored in *X_reject*.

The algorithm for rejecting existence violations is simply to discard rows in *X_staging* which have a null value on a column with a not null property, and send them to *X_reject*. Suppose *X* has a column [Col not null] with a not null property, the SQL is as follows:

[Example SQL]

The algorithm for detecting uniqueness violations has two scenarios. Suppose *X_staging* has a column [Col unique]. The possibilities are:

1. The duplicates exist in *X_staging* when two rows have the same value on [Col unique].
2. The duplicate happens when *X_staging* applies an upsert that creates a value in [Col unique] that already exists in *X*, but not if that same row swaps its value to another one to avoid a duplicate.

In these cases, all but one of the rows should be sent to *X_reject*. Example SQL follows.

[Example SQL]

The two criteria can also be extended to other criteria such as data type violations. In theory, *any* tailored criteria can be applied to abort a load or reject some rows on a particular table. These two cover many of the problems that a large pipeline encounters. They can be applied by default to every load through automation rather than manual tailoring.

Step 3 - Apply

Once the changes are determined in *X_upsert* and *X_delete*, and invalid rows are sent to *X_reject*, the changes can be applied to *X*.

The first step is to delete the rows in *X* using the keys in *X_delete*. Rather than completely losing the rows, it is important to send the deletes to a history table so that downstream processes know which rows have been deleted. In some platforms, this can be done automatically. In this example, the table to hold the old records is *X_history*. The SQL is as follows:

[Example SQL]

Then the upserts should be applied. The rows to be inserted from *X_upsert* into *X* are identified by loading *X_upsert* where [Is new row] = 1. The rows in *X* to be updated using values from *X_upsert* are identified by [Is new row] = 0.

As with deletes, the old values in *X* that are to be updated should be sent to *X_history* rather than being overridden and lost. Keeping these rows is important for incremental processing downstream.

This step of applying changes should also add architectural columns for change datetimes. They are:

- [Row insert datetime] — when was the row, defined by the primary key value, first inserted into the table.
- [Row update datetime] — when was the row, defined by the primary key value, most recently updated with a new value. Because of the *Check* step, these are always genuine changes.
- [Row delete datetime] — when was the row, now in *X_history*, deleted from *X*. This could be because of a full delete by its primary key or because it was overridden by an update.

Some platforms calculate one or more of these automatically.

[Example SQL]

By the end of this step, and if there were no other errors, the changes would be applied to *X*. The additional artefacts – *X_history*, [Row insert datetime], [Row update datetime], and [Row delete datetime] – provide information for downstream tables that rely on *X* to incrementally extract from it.

After loading

Cleaning-up and logging are important part of the load mechanics.

Cleaning-up

If there are no errors, the tables *X_staging*, *X_upsert*, and *X_delete* do not need to be retained. The rows in *X_history* can also be purged after a while, once they have informed incremental processing downstream. In Delta table context, the history rows are the tombstoned rows, and their purge is known as vacuuming.

If there are errors, then the intermediary tables and *X_reject* need to be retained for troubleshooting. It is important to ensure they are secured in the same way as *X*, and do not accidentally cause unintended data leakage.

Logging

In addition, it is essential to log the table's bookmark, and highly valuable to log the change statistics.

The table's **refresh bookmark** is the *datetime of the start of the load, if the load successfully completed*. Suppose the table *X* draws data from *A*, *B*, and *C*. The refresh bookmark of a load defines when *X* started reading from its source data, and any records that appear in *A*, *B*, and *C* after that datetime have come afterwards. When *X* is incrementally extracted in the next batch, the updated records in the source tables are easily identified. This is explained in greater depth in *Tracking changes*.

[SCREENSHOT]

The *Check* step calculates the number of rows to be inserted, updated, and deleted. It is only a small additional effort to log these numbers. While not essential, they form a rich history to understand a table's evolution and can be valuable for troubleshooting. Since they are already calculated as part of the load, it is easily worthwhile to log them.

[SCREENSHOT]

The success, or failure and associated failure messages, should be logged.

[EXAMPLE SCREENSHOT]

Analysis

The full three steps create five extra tables in addition to *X* — *X_staging*, *X_upsert*, *X_delete*, *X_reject*, and *X_history*. It also creates three extra columns — [Row insert

datetime], [Row update datetime], and [Row delete datetime]. The logic is also much more complex than simply dropping and replacing the table *X*.

In a simple environment with five or even fifty tables, all the additional work may be unjustified overhead. In a moderately sized pipeline, the benefits offered by the extra work outweigh the cost. Past a certain point where errors and resource become costly, they are non-negotiable.

From the view of efficiency, the row-by-row check for genuine change seems exorbitant. But in a table with millions of rows and only a small fraction changing in a load, it is much faster to check for changes and apply them than to load all rows and perform any re-indexing.

More importantly, checking each row and updating [Row update datetime] only for rows with a genuine change informs downstream tables how to respond. Without such a check, all downstream tables will need to reload all rows in every batch.

Seen in a different way, the data warehouse has responsibility to keep track of every single change that happens within it. The row check is the foundation for this tracking.

From the view of stability, the row-by-row check plays an important role in stabilising a pipeline of incremental extract by eliminating noisy changes. Incremental extract typically works by selecting candidate change rows through a change detection column. If an external factor causes the change detection column to shift all its value but none of the other column values changed in the row, this process of row-by-row checking eliminates these nil changes and informs downstream tables not to incorrectly react.

To rephrase, a pipeline of incremental extract relies on change detection columns, but when one such column goes wrong, the effect can be disastrous. Instead of relying solely on the change detection column, it should be used as a first pass, while allowing the *Check* step to do a second pass to eliminate any noise if things go wrong.

Finally, the *Check* step improves stability and fault tolerance by limiting the impact of abnormalities or errors. To take advantage of the fault tolerance, tables must not have unstable columns that change frequently for no informational reason, or may suddenly change on many rows, such as on a switch to new year. Examples of these are discussed in the *Fault tolerance* chapter.

On a specific table, the additional work may seem excessive. But doing the work consistently as a pattern is the backbone of a sophisticated pipeline by monitoring and controlling row changes. Investment in automation or appropriate technology ensures that the additional logic is applied by default, with no development cost to the engineer.

Load stack

The chapter *Load mechanics* look at the details of how a single table is loaded. This chapter looks at how load across multiple tables can be orchestrated across the pipeline.

In a data pipeline, tables need to be loaded in the correct order. Even with a small number of tables, such as twenty to fifty, their dependencies can become difficult to manage. Poor implementation, such as manually coding their load order, leads to an unmaintainable pipeline that is hard to debug.

In addition, loading tables in parallel is an easy way to reduce the total amount of time to process a large pipeline. Even a four-fold reduction may mean the difference between meeting the start of the business day or not.

Load orchestration is the task of ensuring the loads in a pipeline happen correctly, always in order, and in parallel if computational resources are available.

There are two main ways of orchestrating a load:

- A centralised approach where a main process *assigns* work to workers
- A decentralised approach where workers *grab work* from a queue

In the centralised approach, a software orchestrates other server jobs to load tables in the right order and manages the parallelism. It is often difficult for data engineering teams to create such software at scale and requires the purchase of a vendor product.

On the other hand, the decentralised approach in this chapter is easy for any team to implement. All it takes are:

- An accurate record of pipeline lineage as metadata, that is, how one table draws data from other tables in the pipeline.
- The ability to load one table at a time, that is, there are no procedures or functions that load two tables in an entangled fashion.

The decentralised approach achieves the aim by maintaining a queue of work, making the queue visible, and allowing workers to grab from the queue. We call this queue the **load stack.**

Load stack and load candidates

The load stack is a list of tables to be loaded. Each row represents a table to be loaded and annotated with two columns: [Is started] and [Is ended]. These indicate whether the load for that table has begun or finished. If [Is started] = 0, the table has not yet started loading and needs processing.

[Screenshot]

But not every table with [Is started] = 0 is ready to load. Loads must follow the correct order. A table is ready only when all the tables it depends on have finished loading, a concept known as **execution readiness**. These are the **load candidates**.

Identifying the load candidates requires a list of each table's load dependencies. That is, the tables which directly feed into a table through a load.

[Screenshot]

It is instructive to visualise the dependencies in a directional graph. Each arrow shows a direct feed from one table into another.

[Screenshot]

By tracing dependencies upstream, the full set of prerequisites for each table can be found. These are the **expanded dependencies** of a table.

[Screenshot]

The load stack and the expanded dependencies are all that are needed to identify the load candidates, that is, the tables that are ready to load.

A table is a load candidate if there are none of the tables in its expanded dependencies that have not ended. That is, there are no dependencies where [Is ended] = 0. This logic also works for tables with no dependencies at all, since their expanded dependency list is empty and therefore contains no unfinished loads. When a table has no unfinished loads in an upstream table, it is a candidate to start loading.

This logic can be exposed as a view. The view shows the current list of load candidates and can be queried at any time to find which tables are ready to load.

[SQL and SCREENSHOT]

Using the load stack

The load stack table and the load candidate view can be used to implement a parallel load using the following algorithm.

Step 1.

At the beginning of the load, populate the load stack and set [Is started] and [Is ended] to 0 for all rows. This is called refreshing the load stack and may require deleting existing rows.

Step 2.

Start any number of workers. These could be functions in sub-processes or stored procedures in independent sessions. Each worker continuously polls the load stack and continues polling until there are no more tables with [Is started] = 0.

If there are still tables to be loaded, the worker queries the load candidate view to pick up a table that is ready to load.

If the view returns a table, the worker

1. sets [Is started] = 1 for that table in the load stack. This is claiming the table for loading,
2. invokes the function or procedure to load the table, and
3. sets [Is ended] = 1 when the load finishes or is aborted due to an error.

If the view returns no tables, the worker pauses for two seconds before polling again.

The worker exits when there are no more tables in the load stack with [Is started] = 0.

Claiming from the load stack must be done in a way that prevents multiple workers from claiming the same table. This can be achieved by wrapping the read from the load candidate view and the update to the load stack in a transaction. This avoids race conditions.

In summary, this is all that is necessary to implement parallel load:

- The load candidate view, informed by expanded dependencies, ensures that a table is only offered for loading if it has no upstream tables still loading. This guarantees correct load order.
- Multiple workers can pick up jobs as soon as they become available.
- Transactions and locks around the claim step ensure that workers do not clash when selecting the same table.

While these steps are sufficient for parallel loading, it is useful to log additional columns.

- Start and end datetimes for each table in the load stack.
- A single [Workflow ID] assigned to every table in the load stack when it is refreshed. This ID represents the batch of load.
- A [Worker ID] generated by each worker, recorded against the table to show how jobs are distributed.

Advantages

The advantages of the load stack are numerous.

No special code for parallelism

There is no special code for parallelism. The logic looks identical whether one worker or many are used. The primary purpose of the load stack and the load candidate view is to ensure that tables load in the correct order. The ability to load multiple tables simultaneously is a natural side-effect of the design. The only difference between serial and parallel load is the number of workers started at a given moment. This does not

need to be known to any central orchestrator. In fact, the load stack does not even track how many workers are active.

Unlimited and dynamic workers
The number of workers is unlimited and can be turned on or off at any time. Many parallel load algorithms decide on the number of workers at the beginning and launch them accordingly. The load stack approach allows the platform engineer to respond to server resources in real time by adjusting the number of workers. Only minor changes are needed to allow workers to exit the polling loop early.

Greedy and balanced work assignment
The algorithm is well load-balanced by default. It uses a greedy approach to assigning work. As soon as a load candidate is ready, a worker picks up the job. There are no situations where a table is ready to load and a worker is available, but the load does not happen. While this does not guarantee optimal load time, it minimises idle workers. For further improvements, the load candidate view can be adjusted to return the next table based on a more fine-tuned balancing logic.

Partial pipeline support
It is easy to load only a portion of the pipeline. The expanded dependencies show exactly which tables are upstream from any given set. Loading only part of the pipeline can be achieved by populating the load stack with just those tables. This allows a subset of the pipeline to run more frequently or on a different schedule than the full pipeline, without needing changes to a central orchestration plan.

Mid-load interference
Since load is controlled by making tables available on the load stack, the approach supports mid-load interference by editing the load stack. For example, tables can be put back on the load stack during a load by setting [Is started] = 0 for those tables. This supports error correction and retries mid-load.

Increased load frequency

Most warehouses run loads in a daily batch, but some business needs require more up-to-date information that could be hourly or minutes latency, demanding a higher load frequency.

In practice, higher load frequencies are only practical if each of the table in the pipeline is incrementally processed. It is unlikely that every table in a large warehouse would be implemented this way. The load stack makes it a possible to load any portion of the pipeline. This means that it is easy to select a path within the overall pipeline for increased load frequency while allowing the rest to happen in usual batches.

In the most demanding case, editing the load stack mid-load can be used to implement continuous loads. For example, workers start to process the pipeline, while another

process periodically resets one or more tables by setting [Is started] = 0 on the load stack. This puts the tables back on the queue for continuous processing.

The load stack allows a data engineering team to treat low and high frequency loads with the same orchestration. A path within a pipeline where all the tables are efficient can be loaded more regularly, increasing the frequency from once a day to multiples times in an hour, and ultimately to run continuously. It only depends on the investment in making each of the table in the path to process within the required response time. This graduation allows “just-enough” investment to meet frequency targets. This gives teams a continuous-load option before considering the need to adopt dedicated streaming platform.

Cross-technology orchestration

The load stack supports orchestration across multiple technologies. While many pipelines run on a single stack, complex problems may require different technologies for different parts of the load. The usual solution is to purchase orchestration software with connectors for each technology. The load stack reverses this. The only requirement is that the central load stack table is visible to all technologies. There is no need for a central worker to have connectors. For example, the load stack and load candidate view may be implemented in a SQL database, with claiming done through a stored procedure. As long as other technologies, such as Spark or other SQL warehouses, can connect via ODBC, they can participate. The load stack enables cross-technology orchestration with minimal requirements.

Conclusion

Load orchestration is the task of ensuring tables in a data pipeline load correctly with respect to their dependencies, even when doing so in parallel.

It is easy for load orchestration to go out of hand, creating pipelines that are opaque and impossible to maintain. When the pipeline becomes complex, some organisations may purchase orchestration software and additional servers to host them.

The load stack approach is a simple alternative. It does not require additional servers or software. Only minor amendments using tools already available to a data engineer are needed. The only requirements for the data engineer are to capture each table with its direct dependencies in a metadata table, and to modularise code so that each table can be loaded independently. The load candidate view computes execution readiness by consulting the state of the load stack to know what jobs have been complete and comparing it to the metadata table of dependencies. This ensures the load happens in the correct order no matter the level of complexity in dependencies, and no matter the number of workers, even in the absence of a centralised orchestrator.

Compared to other common approaches towards orchestration, the load stack is a queue-based coordination adjusted for lineage-awareness. At any given point, it uses

the actual state of upstream tables to determine readiness of tables to load. This allows the load to respond immediately to changes in circumstances, something that is not easy in other approaches.

It is technology-agnostic, lightweight, and easy to adopt — which makes it ideal for any team to implement with low investment cost.

Load dependencies

One of the biggest factors impacting efficiency and stability is the use of dependencies in a pipeline.

Dependencies occur when a materialised table selects from other tables for its load. This is a **load dependency**, because the materialised table needs to be loaded in the pipeline. Load dependencies propagate information from downstream.

Dependencies also occur through views. Views also propagate information, but they do so indirectly by propagating *logic* rather than persisted data. We distinguish them as a **view dependency**.

Load dependency is the focus of this chapter. Whenever the word "dependency" is used in this chapter, it refers to load dependency.

Dependencies improve efficiency because they allow reuse rather than recalculation of the same piece of information. In this sense, they also improve quality by reducing the risk of logic becoming out-of-sync.

However, dependencies can harm stability because failure in one table directly propagates downstream, sometimes leading to domino effect of issues.

Moreover, changing the business logic of a table early in the pipeline theoretically requires all downstream tables to reload, in case the change affects their results. In theory, it is possible to isolate the impact by examining the logic change. In practice, this is too error-prone, and it is safer to reload downstream tables. This may lead to a situation where small changes upstream cause large changes downstream. In this scenario of logic change, dependencies can be harmful to both efficiency and stability. They are inefficient because large amounts of processing may occur with little change in outcome. They are unstable because during the reload, many tables become temporarily empty as they are reloading.

Ultimately, at every turn the data engineer must choose between re-calculating or re-using existing information. This is a decision based on trading efficiency and stability.

Criteria for dependency

There are three criteria to consider when committing to a dependency. The dependency should be:

- **Valuable** – the information is worth selecting and not easily recomputed.
- **Targeted** – the source table is narrow, rather than wide with miscellaneous columns.
- **Stable** – small changes in the source should not cause massive changes downstream.

Valuable

Using dependencies to propagate information is convenient in the short term. However, the issues that come with a heavy chain of dependencies often take time to surface. This can lead new data engineers to use dependencies too casually and not notice the problem until it is too late.

The first step in managing dependencies is to ensure that the selected information has enough value to justify the dependency.

For example, the data engineer should avoid bringing in "convenience" columns from other tables. A common case is reading the country name from the reference country table simply for display or debugging purposes early in the pipeline. Instead, the fact table may only need the country code, and the join to the reference table can occur later in the presentation layer through a view.

Another method is to recalculate where it is simple. For instance, computing the first of month from a date column is preferable to looking it up from a calendar table. This may involve repeating simple logic, but it avoids forming a dependency. Similarly, using a standardised default value for unknown references can reduce unnecessary joins. For example, if the surrogate key for unknown values is always "1", this number can be hardcoded in the transaction table with little risk, avoiding the need to select from the reference table to find the surrogate key value by name.

Targeted

Dependencies should be targeted — selecting just enough for their purpose. When dependencies are not well targeted, more dependencies are formed than are actually useful, thus diluting the value of the dependency relative to its cost.

The smallest unit of dependency is a table. That is, when a table selects a column from a source table, the entire source table becomes a dependency. If many columns are packed into one table, then any table that needs just one of those columns must depend on the whole. This leads to a concentration of dependencies on that single table. Wide tables with miscellaneous columns not only create tangled dependencies but also reduce clarity of information flow.

As an exaggerated example, if there were a table called *Cake.Everything*, and multiple tables selected from it, then any update to *Cake.Everything* would require updates to all

downstream tables. It would also be unclear what information about cake is being passed downstream.

Meaningful fragments are excellent for breaking down dependencies to ensure they are more targeted. Continuing the example, *Cake.Everything* can be split into well-defined fragments such as *Cake.Sales*, *Cake.Milestones*, and *Cake.Quality*. Selecting from these fragments helps target the dependency and makes it clearer what information is being selected.

It is often better to start a new fragment than to add a new column to an existing table — especially if the new column introduces a different concept. For example, if there is a *Cake.RefCalendar* table used to describe dates, and there is a need to include sales seasons, it may be tempting to add a column like [Is sales season]. This is problematic because it increases the weight of dependency on *Cake.RefCalendar* when the sales season is needed for analytical computation. Instead, it is better to create *Cake.RefSaleSeason* with a primary key [First date of week] to indicate that sale seasons operate at the week grain. *Cake.RefSaleSeason* would be a narrow table with just enough information to describe the sale season. Downstream tables would select from *Cake.RefSaleSeason*, offloading the dependency from *Cake.RefCalendar* and making it clear that the table is interested in sale seasons. *Cake.RefSaleSeason* would relate to *Cake.RefCalendar* via a foreign key on [Calendar date].

Stable

Stability means ensuring that small changes in the source do not cause large, unintended changes downstream.

A common example of instability is when a transaction table selects the country name from a reference table. If the name of a country is updated, that single change may trigger millions of downstream updates. This kind of ripple effect can be costly and difficult to manage.

Where possible, dependencies should be stable. Where not possible, the risks should be mitigated. The two main sources of instability are **row-wise instability** and **column-wise instability**.

Row-wise instability occurs when a single row drives many downstream rows. This is typical of reference tables. A small change in a reference table can affect large volumes of transaction data. Metadata-driven loads are another example. If a pipeline reads a metadata table to decide what to load, an error in that metadata can cascade into widespread changes.

Changes tend to be amplified when a dependency selects one row and applies it across many rows — for example, reading a small table into a large one. Small tables are often a source of instability for this reason.

There are several ways to mitigate row-wise instability. In the country name example, the name may not need to be denormalised into the transaction table at all. It can remain in the dimension table for presentation, while the fact table stores only the country code. More generally, moving information from transaction tables to reference tables, such as adding computed columns to reference tables, and allowing the reference tables to carry the information load is best practice. Another option is to defer lookups until later in the pipeline or perform them only through views.

Column-wise instability occurs when the source column itself is volatile. For example, selecting [Update datetime] into a downstream table introduces risk. A routine system update that refreshes all timestamps may seem trivial to the application but can cause widespread downstream updates.

To reduce this risk, a polling table can be used. This technique is explained in the chapter *Tracking changes*.

Another example is a column like [Is archived]. A digital system may perform bulk updates using this column. From the system's perspective, the change may only affect what appears in the UI, but for the data pipeline, it can have a significant impact.

Column-wise instability is less common than row-wise instability, but it is important to monitor for both.

The case of surrogate keys

Surrogate keys act as stand-ins for the primary key of a table. Most commonly, they allow a composite primary key made up of multiple columns to be replaced by a single integer. Surrogate keys may be created by source systems, but they are often introduced by the warehouse as part of the data pipeline.

Surrogate keys are necessary in a warehouse for several reasons. First, and most importantly, Power BI supports only single-column relationships. If the primary key consists of multiple columns, a surrogate key is required to implement the relationship. Surrogate keys also simplify Power BI measures that rely on distinct counts, making implementation more straightforward.

Second, multiple-column keys can be clumsy to carry. This is especially true for storytelling dimensions, where the primary key may consist of several binary columns — sometimes more than ten. It becomes cumbersome to record all ten columns on each transaction table and join on all of them. The same issue arises with type II keys such as [Country code], [Start datetime]. A lookup to such a table involves an inequality join on the validity period [Start datetime] and [End datetime], which is error-prone when repeated across the code base.

Third, some primary keys are slow to join on. This can happen with type II keys due to their complex predicates, or when the primary key uses data types that are not conducive to performant joins, such as long strings.

For these reasons, it is often desirable to retrieve a surrogate key. This involves looking up the table using its primary key and bringing in the surrogate key column. In this way, surrogate key retrievals can be a valuable dependency.

However, surrogate keys can become magnets for dependency. They are particularly unstable. If the surrogate keys are regenerated, because the table needs to be reloaded for amending a logic, all downstream key values reshuffle.

There are two ways to manage surrogate key dependencies.

The first is to defer the lookup to later stages of the pipeline, perhaps only during the final load into the Power BI semantic model. If the original business key consists of only two columns and join performance is acceptable, this deferral may be reasonable. However, if many downstream tables rely on this key and the data types are not ideal for joins, it may be better to accept the cost and swap out the keys earlier in the pipeline.

The second approach is to compute the surrogate key rather than look it up. For example, in a storytelling dimension with five binary columns as the primary key — [Is A], [Is B], [Is C], [Is D], [Is E] — the surrogate key can be calculated by treating these as a base-two number and converting it to base-ten using a simple formula:

[SK] = 1 + 2^[Is A] + 4^[Is B] + 8^[Is C] + 16^[Is D] + 32^[Is E]

Using this formula, it is possible to derive the correct surrogate key value without performing a lookup.

Whether using this or another formula, if the computation is simple, calculating the surrogate key on the fly can provide the benefits of a surrogate key without forming a dependency.

Views as an alternative

Views are also a form of dependency that can propagate information. However, they are not a direct substitute for load dependencies. Load dependencies propagate information itself, while views propagate logic. For example, when two tables are joined in a view, it is the logic of the join, not the result of the join, that is passed downstream. This makes view dependencies and load dependencies fundamentally different.

However, views can be effective as an alternative in a range of scenarios, especially when the dependency is not stable. For instance, instead of denormalising a country name into a materialised table, a view can perform the same join. In this case, a change to the country name in the reference table does not trigger any update operations

downstream. However, when the user queries the view, they retrieve the updated name. This is particularly useful in parameterised reports, where users look up sections of data at a time and it is not necessary to have the full set materialised in advance.

In theory, a server with infinite resources could support an entire pipeline built from views. This would eliminate the kind of instability caused by materialised dependencies. However, views can also amplify instability by instantly propagating any errors downstream. Materialised tables, by contrast, offer a buffer of stability for reporting.

Another limitation of views is that they do not track changes. As explained in the *Load mechanics* chapter, the *Check* step of a load verifies whether a row has genuinely changed. This allows downstream workloads, such as machine learning predictions, to respond appropriately. Views do not support this type of row-level change detection.

Because views propagate logic rather than information, they also lose historical data when source systems delete older records. This can be problematic for pipelines that require long-term persistence.

Views can be an effective alternative to load dependencies under certain conditions. They are suitable when query performance is adequate, when instant propagation of changes (including potential errors) is acceptable, when long-term persistence is not required, and when row-level tracking of changes is not needed for downstream processes.

Conclusion

A good pipeline should contain a healthy level of dependencies. This is a sign that the data engineer is adding value to raw data through transformation, aggregation, and storytelling. A pipeline with almost no dependencies often reflects a garbage-in, garbage-out approach, or complex logic buried in a few scripts.

As a rule of thumb, a source system with 10 to 20 raw tables may result in 5 or 6 layers of dependencies. In larger systems with 30 to 50 tables, especially when integrated with other systems, the transformation pipeline may easily reach 10 or more layers. In the site where this work is done, there are 10000+ tables, 15000+ dependencies between them, amounting to 25+ layers of dependencies.

s 22(1)(a)(ii)

Dependencies are an effective tool for propagating information and achieving efficiency. They allow the data engineer to build reusable blocks of insight, articulate business meaning, and pass them for use. But they must be used with care. Each dependency increases entanglement and can reduce stability. A data engineer must be deliberate in committing to a dependency by selecting only those that are valuable, targeted, and stable.

Tracking changes

The aim of information efficiency is to process only what has changed. If there were no records changed after a table is processed, then ideal is have spent zero processing time it. In practice, even when there are no changes in input, a load still takes time to process.

Every unnecessary scan consumes resources without adding value. Efficiency is achieved when downstream tables can reliably extract only the records that have changed and ignore the rest.

Doing this well requires systematic change tracking. Without it, pipelines fall back bespoke methods for tracking changes. This leads to errors in incremental processing, periodic full reloads, and fragile recovery from disruptions. A robust pattern avoids these pitfalls.

The following is a simple approach illustrate the concept of change tracking, before unpacking the details of a more robust approach.

Simple approach

The most basic approach is to include an update datetime column from the source system and filter on it. This works when the source table deletes do not occur, and the target table mirrors the source one to one.

Example scenario

- Source table Raw.Event with columns [PK1], [PK2], [Col1]...[Col15], [Update datetime]
- Target table Curated.Event with [PK1], [PK2], [Col1], [Col2], [Update datetime]

Incremental extract logic would be as follows:

```sql
-- step 1: find the latest update in the target
set @latest_update_datetime = (
    select max([Update datetime]) from Curated.Event
);

-- step 2: pull only new rows from the source
select *
from Raw.Event
where [Update datetime] > @latest_update_datetime;
```

While straightforward and intuitive, this approach suffers from a few problems.

First, it is not scalable to complexity. Additional source tables will break the logic because it is not practical to add the update datetime from each source onto the target table. It is possible to create a complex compound datetime, but it would be error prone.

Second, it is not stable. If a system update shifts all values in [Update datetime], the pipeline will reprocess every row, causing a blow out in processing time for updates, and triggering similar issues downstream.

Third, the approach is not scalable to continuous load. For example, if *Curated.Event* filters *Raw.Event* for *rare* events, then the maximum update datetime on *Curated.Event* will not represent latest processing time of *Raw.Event*. The simple approach will force a re-scan that goes too far back. For a continuous load scenario, this will be prohibitive.

A more scalable and reusable pattern is to separate the time-tracking artefacts from the data content itself.

Refresh bookmarks and polling tables

As explained in *Load mechanics*, as each table is processed, the pipeline logs the **refresh bookmark** of a load. That is, the *starting datetime* of the load of each table is logged *if the load succeeded*. The refresh bookmark for all tables in the pipeline can be stored in one table. Each row of this table is one table that was loaded, together with the starting datetime of that load.

Results Messages

	Schema	Name	Bookmark datetime
1	_	ColumnDataType	2025-10-20 22:57:13.4696455
2	_	ExpandedDependency	2025-10-20 22:57:14.6578606
3	AA	AccreditationStatus	2025-10-17 16:36:40.2663248
4	AA	AccreditedPerson	2025-10-17 16:36:38.4207202
5	AA	AccreditedPersonArrangement	2025-10-17 16:54:45.4743562
6	AA	AccreditedPersonStatus	2025-10-17 16:36:40.9069570
7	AA	Application	2025-10-17 16:54:45.7564726
8	AA	ApplicationAudit	2025-10-17 16:36:41.2038441
9	AA	ApplicationClockFreeze	2025-10-17 16:22:07.0639376
10	AA	ApplicationTask	2025-10-17 16:10:58.1659837
11	AA	Contact	2025-10-17 15:52:04.2453583
12	AA	RefTask	2025-10-17 16:09:10.8291377
13	ABR	Agency	2025-02-21 04:53:09.5069865
14	ABR	AuthorisedContact	2025-02-21 06:14:50.4208592
15	ABR	BusinessAssociate	1900-01-01 00:00:00.0000000
16	ABR	BusinessLocation	2025-02-21 03:40:02.8040690
17	ABR	BusinessName	2025-02-21 04:12:47.0192706

This bookmark defines the cut off for what records in a source has fully made it to the processing of a target table. Any records that have come to the source table after this bookmark datetime would need to be considered in the next batch of processing. If the table load aborted because of a fault, then the bookmark should not be logged so that the next batch can resume from last success point.

An advanced note. If the source is continuously loading, then some stray records may still come into the source table after the bookmark datetime, but before the processing has complete. However, there are no guarantees. Consequently, either the incoming source needs to be frozen, or the next batch should still resume from bookmark and reprocess these late records.

The query to retrieve the latest bookmark for a table is simple:

```
declare @refresh_bookmark_datetime  datetime2(7);

--  step 1: look up the refresh bookmark datetime for Curated.Event
--  this is when the previous load started
select @refresh_bookmark_datetime =
(
    select [Bookmark datetime]
    from   _.RefreshBookmark
    where  [Schema] = 'Curated' and [Name] = 'Event'
);
```

On the other hand, keeping track of source changes depends on whether the source table was created by the pipeline or whether it was raw data table from external source.

In the former case, the tables created by the pipeline have reliable update datetimes created by the process in *Load mechanics* – [Row insert datetime], [Row update datetime] and [Row delete datetime]. Since these are created by the pipeline itself, they are comparable with the refresh bookmark and can be compared directly for understanding what records have come in after the latest bookmark.

However, for tables that were not created by the pipeline, even though they may have a column called [Update datetime], there may be a substantial lag between that value versus when the row made it to the database. Hence, they cannot be used directly.

To distinguish these two cases, we would say that the update datetimes in pipelines are **in-sync** while the update datetimes from source systems that are loaded in a separate process are **out-of-sync** with the data pipeline.

If the datetimes are out-of-sync with the refresh bookmark, they cannot be directly compared. Instead, just as the target table's state of processing can be tracked by a separate table, the source table's state of processing can be tracked by a separate **polling table**.

A polling table is any table that can be consulted to know whether and what portion of data needs to be refreshed. For a polling table to be effective, it needs to be structured so that it can be queried rapidly – much more rapidly than the time to process the full data. Ideally, this query should be close to zero time to support continuous load.

Following the example, direct comparison of *Raw.Event*[Update datetime] with the latest bookmark in *Curated.Event* is not possible. There can be an unknown lag between the update timestamp and its arrival in the database. The only safe assumption is that the datetime increases monotonically.

A polling table provides a way to map the pipeline's refresh time to the latest update datetime observed in *Raw.Event*. Suppose this table is called *Raw.Bookmark*, and is loaded using an append-only process. Each appended row records the maximum update datetime from *Raw.Event* at the time of the refresh.

```
1  insert into Raw.Bookmark ([Refresh datetime], [Bookmark datetime])
2  select sysutcdatetime(), max([Update datetime])
3  from   Raw.Event;
```

The columns of *Raw.Bookmark* are:

- [Refresh datetime] — the datetime of the check as the row is appended. This datetime, since it is computed by the load, is part of the pipeline and thus in-sync with the refresh bookmarks.
- [Bookmark datetime] — the maximum of *Raw.Event*[Update datetime] as at that time of refresh. This datetime, coming from the source and later loaded, is out-of-sync.

This table maps the [Update datetime] column, which is out-of-sync with the pipeline, to a column, [Refresh datetime], that is in-sync with the pipeline. This mapping makes it possible to compare changes to the incoming data tracked by the source's [Update datetime] with processing events, which are tracked against the pipeline's record of datetimes.

In theory, refreshing this table infinitely often would allow the pipeline to translate its own time into the update datetime of *Raw.Event* at any point. In practice, the refresh frequency aligns with the pipeline cadence. There is no benefit in refreshing the table when *Raw.Event* has not changed. In addition, *Raw.Bookmark* can contain the bookmarks for multiple tables, such as *Raw.Event, Raw.Event2, Raw.Event3* rather than creating one table for each.

At the start of processing *Curated.Event*, the polling table *Raw.Bookmark* allows the pipeline to fetch the maximum bookmark of *Raw.Event* as of *Curated.Event's* previous

batch start. Any row on *Raw.Event* whose [Update datetime] represents data that have come after the previous batch start and eligible for processing.

```
declare @refresh_bookmark_datetime  datetime2(7);
declare @latest_process_datetime  datetime2(7)

--step 1: look up the refresh bookmark datetime for Curated.Event
--this is when the previous load started
select @refresh_bookmark_datetime =
(
    select [Bookmark datetime]
    from   _.RefreshBookmark
    where  [Schema] = 'Curated' and [Name] = 'Event'
);

--step 2: obtain the latest processed update datetime captured by that refresh
select @latest_process_datetime =
(
    select max([Bookmark datetime])
    from   Raw.Bookmark
    where  [Schema] = 'Raw' and [Name] = 'Event'
        and [Refresh datetime] = @refresh_bookmark_datetime
);

--stage 3: fetch rows whose source update is newer than what the last batch captured
select  e.*
from    Raw.Event as e
where   e.[Update datetime] > @latest_process_datetime;
```

In summary, tracking records that have changed in a source table since the last process depends on two components:

- **Tracking the target**. A bookmark table for the pipeline that records the processing datetimes of each table whenever a load completes successfully.
- **Tracking the source**. If the source table has an [Update datetime] column that is synchronised with the pipeline, this column can be used directly to retrieve updated rows. If not, a polling table is required to map the source table's update datetime back to the pipeline's datetime for comparison.

With these components, it becomes possible to identify changes in the source relative to the target's last processed state.

Change detection columns

The previous example assumes that the source table *Raw.Event* has a reliable [Update datetime] column for change detection. This is not always the case.

In the ideal scenario, the change detection column is an architectural column from the source system. Examples include server constructs, trigger-managed datetimes, or built-in datetimes from commercial products that can be trusted. This is also true for pipeline tables loaded as described in *Load mechanics*.

Less ideally, the change detection column is managed by the source application through application logic. These columns are generally reliable but can be prone to developer error or fail to capture direct database changes.

Finally, there may be no change detection columns at all. In this case, it is not possible to determine whether a source row has been updated. However, it may still be possible to identify inserted records by performing an anti-join between the source and target tables. This approach is useful for append-only loads such as building of hub tables.

Tracking deletes

Polling tables can be used for tracking updates and inserts. Tracking deletes is equally important. However, this can be far more difficult because, in the case of deletes, rows disappear completely from the table itself.

In the ideal scenario, deletes are tracked in the source through architectural artefacts. Some database technologies implement default history tables or delta logs. These history tables allow reliable tracking of deletes, provided they include a change detection column to indicate when the delete occurred.

Less ideally, some applications maintain dedicated business audit tables that track changes, including deletes. These tables can be highly reliable, though they are often difficult to query.

In the worst case, there is no change tracking. Rows can disappear without a trace. The only option is to compare the source and target using an anti-join on the primary key.

The role of the *Filter* step

Two themes recur, whether for tracking upserts or tracking deletes:

- Source update datetimes that are synchronised with the pipeline's process datetimes are trivial to query.
- Architectural artefacts, rather than business artefacts, are the most reliable way to track inserts, updates, and deletes.

When either of these is absent, tracking changes becomes complex or even impossible.

This is one of the reasons why pipelines begin with a *Filter* step that keeps transformation minimal. The filter step is designed to be the first interaction with the source data. On first arrival of the data, the pipeline adds the architectural artefacts required for change tracking. Doing this once at the start allows the rest of the pipeline to apply transformations incrementally with reliable change tracking.

The optimal case is if the incoming data can be incrementally extracted using polling tables and the filter step adds the architectural columns for change-tracking. This effectively converts the source change detection columns, which may be out-of-sync

with the pipeline, to change detection columns which are in-sync the datetimes tracked by the pipeline.

The *Filter* step, which is a simple extract of the necessary rows and columns with minimal joins or aggregations, may look like a step with low value-add. However, this step is serving as a critical foundation for an efficient pipeline because proper load mechanics computes change-tracking artefacts to support efficient downstream response.

Conclusion

An efficient pipeline requires knowing exactly what changed in the source so that it responds only to those records. These changes include inserts, updates, and deletes.

A simple approach to tracking upserts is to include the source change detection column in the target table. However, this can be problematic for non-trivial extracts and is at risk of instability. Instead, bookmarks to track the target and polling tables to track the source are reliable patterns for tracking changes relative to each other.

Deletes are much harder to track without additional artefacts or audit tables. If these artefacts are not available, a full comparison of each batch is necessary to identify changes for upserts, deletes, or both.

Given the complexity of tracking sources and the fragility of relying on other systems to maintain reliable architectural columns, it is important that each pipeline begins with an uncomplicated extract from the source data. The *Load mechanics* process annotates this extract with reliable change detection artefacts for downstream processing.

Responding to change

The previous chapter *Tracking changes* focused on tracking changes in the source. The chapter builds on the previous and addresses the harder task of responding correctly to these changes. This is often known as **incremental extract and load**.

Accurately responding to change can be error-prone. This is because inserts, updates, and deletes upstream do not translate directly into the corresponding changes downstream. The following examples illustrate the possible complexities:

- **Inserts can trigger deletes.** The *Bank.CustomersToFollowUp* is a computed table of customers who have not made recent deposits and require follow-up by the service team. This table takes input from the *Bank.Transactions* table. In this case, an insert into *Bank.Transactions*, such as a new deposit, would trigger a delete in *Bank.CustomersToFollowUp*.
- **Deletes can trigger updates.** The *Bank.AccountSummary* is a computed table of account-level aggregates, such as total number of holders and balance per account. It takes input from *Bank.AccountHolder*, which records all individuals associated with each account. If a joint holder is removed, resulting in a delete from *Bank.AccountHolder*, this changes the holder count and may trigger an update in *Bank.AccountSummary*.
- **Updates can trigger inserts.** The *Bank.GoldCustomer* is a computed table of customers whose account balance exceeds a defined threshold. It takes input from the *Bank.AccountBalance* table. In this case, updating a row in *Bank.AccountBalance* to increase the balance may cause a customer to newly qualify for gold status, triggering an insert into *Bank.GoldCustomer*.

These examples are highly artificial. However, it is easy to imagine the possibilities under general circumstances:

- The presence of **anti-joins** (e.g. not exists) means that inserts upstream can trigger deletes downstream.
- The presence of **aggregation or windowing** (e.g. group by) means that deletes upstream can trigger updates downstream.
- The presence of **filters on derived expressions** (e.g. having) means that updates upstream can trigger inserts downstream.

The following is an illustrative list of correct ways to respond to upstream changes, and a pattern for applying this response to incrementally loading a table.

Analysing the query

Suppose there are two tables *X* and *Y*, and these are used as inputs to a data pipeline. The set up is as follows:

- X has columns [Header ID], [Value], [Status], where [Header ID] is primary key.

- Y has columns [Header ID], [Line number], [Value], [Status] , where [Header ID] and [Line number] is primary key.

And *Y*[Header ID] is a foreign key to *X*[Header ID] and joins on this column.

There could be inserts, updates and deletes on X, as well as on Y. Thus, there are six possible input changes each time. In practice, inserts and updates and can often be considered as a single upsert. Thus, the inputs are:

- upserts from X
- deletes from X
- upserts from Y
- deletes from Y

Suppose the target table is T. The goal is to determine which rows in the target table need to be inserted, updated, or deleted. This is done by computing a driver set of primary keys that might be affected by changes in source tables (X and Y). Inserts and updates can be considered as one. Thus, the goal is to calculate:

- primary keys to upsert on T
- primary keys to deletes on T

The way to calculate these depends on the nature of the query. Some possible scenarios include:

1. A straight select of *X*
2. A filter of *X* on status
3. X inner join Y
4. X inner join Y group by [Header ID] to sum over Y[value]
5. X left join Y coalesce on [Line number] to 0 where Y[Header ID] does not exist
6. Y left join X
7. X left join Y group by [Header ID] to sum over Y[value]
8. X union Y
9. X left anti-join Y

All of these would require a different treatment. These are summarised in the table:

Scenario	Primary key	Approach	Primary keys to upsert in T	Primary keys to delete in T
1. Straight select of X	[Header ID]	Simple case: X changes map directly to T.	Headers from X that were inserted or updated	Headers from X that were deleted

2. Filter of X on status	[Header ID]	Filter condition means updates can add or remove rows.	Headers from X that were inserted or updated and now pass the filter	Headers from X that were deleted, or updated and no longer pass the filter
3. X inner join Y	([Header ID], [Line number])	Join means changes on either side can add or remove pairs.	Pairs for Y rows inserted/updated, plus pairs for X headers inserted/updated with matching Y rows	Pairs for Y rows deleted, plus all pairs for headers deleted from X
4. X inner join Y with group by	[Header ID]	Deleting a Y row can change the aggregate or remove the header.	Headers from X or Y that changed and now have Y rows; include changes in Y, deletes in Y can also change the aggregated value	Headers deleted from X, or headers that now have zero Y rows
5. X left join Y with synthetic line 0	([Header ID], [Line number])	Synthetic line 0 appears only when no Y rows exist.	Pairs for Y rows inserted/updated; plus (Header ID, 0) for headers with no Y rows; plus pairs for new X headers	Pairs for Y rows deleted; remove (Header ID, 0) when Y rows exist; remove all pairs for headers deleted from X
6. Y left join X	([Header ID], [Line number])	T follows Y's lines; X changes matter only if Y lines exist.	Pairs from Y inserted/updated; plus pairs under headers in X that changed	Pairs from Y that were deleted
7. X left join Y with group by	[Header ID]	X controls row presence; Y only changes the aggregates.	Headers in X inserted; headers where values changed due to X or Y changes, including deletes from Y	Headers deleted from X
8. Union of	[Header ID]	Header exists if present in either X or Y.	Headers inserted/updated in X or Y; headers	Headers deleted from both X and Y

projected headers			deleted from one side but still exist on the other	
9. X left anti-join Y	[Header ID]	Row appears only when X exists and Y does not.	Headers inserted/updated in X with no Y rows; headers where Y rows were deleted and now qualify	Headers deleted from X; headers that gained a Y row

These show an array of considerations for two tables. If there are more tables, even up to 10, the analysis can become complicated.

Consider a query that draws from several tables, such as:

```
select
    *
from A
join B on B.[PK] = A.[PK]
join C on C.[PK] = B.[PK] and C.[PK2] = B.[PK2]
```

The first step is to **determine conditions for upsert in the target.**

For each source table, evaluate how changes in that table may influence the result of the query. In general:

- **An upsert in a source table typically results in an upsert in the target.**
  Any insert or update in A, B, or C may alter projected values and therefore requires re-computation.
- A delete in a source table may also result in an upsert in the target.
  This occurs when the query includes:
  - Left joins, where the absence of a row changes a value to a default rather than removing the target row.
  - Aggregations, where the removal of a contributing row modifies a computed measure.

Thus, for each table, the question is:
"Could an insert, update, or delete in this table change the values produced by the query?"
If so, the corresponding key should be included in the driver set for upserts.

The next step is to **determine conditions for deletes from the target.**

In most cases, this occurs only when the tables that governs row presence loses the corresponding row. For example:

- If the query begins with A left join B left join C ..., then deleting a row from A removes the target row. Deleting a row from B or C generally does not remove the target row; it only changes values.
- If the query begins with A inner join B left join C, then deleting a row from A or B removes the target row.

But this can also happen through updates or inserts:

- In the case of A left anti-join B, an insert to A can cause a row to delete.
- In the case of A filtered on a row value, an update to the value can cause a delete.

In summary, upserts arise from any change that affects projected values. Deletes occur only when the base row is removed. Filtering predicates and grouping clauses may cause deletes in a source table to behave as upserts in the target, as they alter aggregates or conditions.

The same reasoning applies to more complex constructs such as unions or set operations, as these are ultimately composed of similar units.

Working through these scenarios can be error prone. Latest innovations in generative AI can provide valuable assistance in this analysis. For example, the table of solutions generated for the 9 scenarios above was generated by GPT-5 using only column 1.

Applying the change

Analysing the query with respect to changes from the source tables informs an incremental extract. Recall from *Load mechanics* that the difference between incremental extract and full extract are:

- The staging table for the load has only a minimal set of records that is much smaller than the full set, but still covers all the records that would need to be upserted in the current batch. This is what makes the load fast.
- Deletes cannot be done automatically by comparing the full set of primary keys between the staging and the target. Instead, it needs to be customised for the query by analysing the impact of changes in source.

The detailed implementation of an incremental extract follows a consistent pattern.

Write query as normal

Begin with the full query that expresses the business logic. This is the reference point for all subsequent steps.

```
select
      A.*
    , B.*
from      Cake.A
inner join Cake.B  on B.[PK] = A.[PK]
```

Fetch the bookmark

At the start of the load, retrieve the bookmark that records the last successful refresh. This value defines the boundary for change detection. The bookmark for all tables loaded by the pipeline should be logged to a central location and easily retrieved.

```
declare @refresh_bookmark_datetime datetime2(7);

select @refresh_bookmark_datetime =
(
    select [Bookmark datetime]
    from _.RefreshBookmark
    where [Schema] = 'Cake' and [Name] = 'T'
)
```

Create minimal staging table

Using the analysis of the query, create a temporary table of keys to upsert to the target.

```
drop table if exists #keys_to_upsert;

-- upserts from A
select
    A.[PK]
into #keys_to_upsert
from Cake.A
where A.[Row update datetime] > @refresh_bookmark_datetime

    union

-- upserts from B
select
    B.[PK]
from Cake.B
where B.[Row update datetime] > @refresh_bookmark_datetime
;
```

In these examples, the driver table has the full key. In practice, it may be a primary key part in the case of multi-columns primary keys.

Attach the driver set as an inner join to the original query. If performance requires, add a clustered index to the driver table.

```
select
    A.*
  , B.*
from Cake.A
inner join Cake.B on B.[PK] = A.[PK]
inner join #keys_to_upsert u on u.[PK] = A.[PK] -- downfilter original query for a smaller staging
;
```

This is now a minimal staging table for incremental load.

Create the delete set

Using the analysis of the query, create a temporary table of keys to delete from the target. This can be used to delete records that should no longer be retained by logic of the query.

Retrieving the records that are deleted would vary depending on how they are tracked. Consider the case where overwritten or deleted records are moved to a history table. Then the examples would be:

```
drop table if exists #keys_to_delete;

-- deletes from A
select
    A.[PK]
into #keys_to_delete
from Cake.A_History H
left join Cake.A A on A.[PK] = H.[PK]
where A.[Row delete datetime] > @refresh_bookmark_datetime -- recently updated or deleted rows
    and A.[PK] is null -- truly deleted

    union

-- deletes from B
select
    B.[PK]
from Cake.B_History H
left join Cake.B B on B.[PK] = H.[PK]
where B.[Row delete datetime] > @refresh_bookmark_datetime -- recently updated or deleted rows
    and B.[PK] is null -- truly deleted
;

-- apply the deletes
delete from Cake.T
from Cake.T
inner join #keys_to_delete D on D.[PK] = T.[PK]
```

This step works even if the source rows were inserted and deleted in between the load of the target table, for example, if the target table failed to load in the interim.

Inexperienced engineers can make the mistake of over-deleting with the idea of re-inserting extra records. This can be tempting because a delete + insert can be easier to calculate than a finely targeted delete + upsert. However, this approach is harmful because it is a source of potential instability. In unpredictable circumstances, this may

trigger the whole table to delete. Instead, deletes should be perfect and remove only rows that no longer satisfy the query's presence rule.

The full script follows.

```sql
-- 1. Fetch the refresh bookmark, which is the point to resume
declare @refresh_bookmark_datetime datetime2(7);

select @refresh_bookmark_datetime =
(
    select [Bookmark datetime]
    from _.RefreshBookmark
    where [Schema] = 'Cake' and [Name] = 'T'
)


-- 2. Determine the upsert driver table
drop table if exists #keys_to_upsert;

-- upserts from A
select
    A.[PK]
into #keys_to_upsert
from Cake.A
where A.[Row update datetime] > @refresh_bookmark_datetime

    union

-- upserts from B
select
    B.[PK]
from Cake.B
where B.[Row update datetime] > @refresh_bookmark_datetime
;


-- query for staging
select
    A.*
    , B.*
from    Cake.A
inner join Cake.B           on B.[PK] = A.[PK]
inner join #keys_to_upsert u on u.[PK] = A.[PK] -- downfilter original query for a smaller staging
;

drop table if exists #keys_to_delete;

-- deletes from A
select
    A.[PK]
into #keys_to_delete
from    Cake.A_History H
left join Cake.A        A on A.[PK] = H.[PK]
where   A.[Row delete datetime] > @refresh_bookmark_datetime -- recently updated or deleted rows
    and A.[PK] is null                                      -- truly deleted

    union

-- deletes from B
select
    B.[PK]
from    Cake.B_History H
left join Cake.B        B on B.[PK] = H.[PK]
where   B.[Row delete datetime] > @refresh_bookmark_datetime -- recently updated or deleted rows
    and B.[PK] is null                                      -- truly deleted
;

-- apply the deletes
delete from Cake.T
from    Cake.T
inner join  #keys_to_delete D on D.[PK] = T.[PK]
```

This workflow offers several advantages.

Uniformity. The same query logic underpins both full loads and incremental loads. The difference lies only in the addition of a driver set join.

Idempotency. The process works regardless of how many times it runs within an interval. If a load fails and is retried, the bookmark ensures correctness. If the load runs more frequently than usual, the outcome remains consistent.

Graceful fallback. If upstream changes touch many rows, say due to a system update, the driver set expands and the incremental load behaves like a full load. The fully load staging table would be compared to the target and only genuinely updated records are applied. There is no explosion of deletes that would cause a blow out in operations.

Best practice workflow

Given the complexity of responding to change, this section outlines a step-by-step recipe for developing and validating pipelines that respond to upstream inserts, updates, and deletes. The workflow balances technical correctness with performance assurance and safe deployment.

Start with a unit test

- Compare row counts between the target table and the expected query.
- Use a datetime filter for performance, but ensure it's independent of the extract datetime.

Simulate data needing update

- Load the table in full, then wait for source data to change to simulate an incremental load.

Build the upsert driver

- As each source table is added, re-run the driver query to check performance.
- Apply indexes if performance degrades.

Test the query filter

- Ensure the inner join between the driver and the original query is performant.

Calculate deletes

- Ensure the delete driver query runs quickly.

Simulate deletes

- Rather than actually deleting, use a select to simulate deletes during development to avoid accidental data loss.

Apply changes and validate Uniqueness, existence, and stability violations

- Apply upserts and deletes.
- Compare the result with a fully loaded copy of the table.
- The unit test should pass.

Zero-load benchmark

- After loading, re-run the upsert and delete drivers.
- They should return no rows and run in near-zero time.
- The time it takes to do this is the fastest the query can ever run.

Run incrementally over time

- Continue loading incrementally over multiple days.
- The unit test should continue to pass.

Conclusion

A transformation query can be seen as a formula that takes inputs and produces output. Information efficiency is achieved when changes in the input are accurately tracked and correctly translated into changes in the output. The difficulty of doing this depends not on the volume of data, but on the shape of the query. Anti-joins, aggregations, filters, and window functions all introduce complexity in determining which rows to upsert and which to delete. This determination should be made by analysing the query through the lens of relational algebra. Modern tools, including generative AI, can assist in this analysis by surfacing the logical structure of the query and identifying the propagation of change.

Once the analysis is complete, the transformation can be converted from full to incremental using a standardised approach. This is to fetch the refresh bookmark of the target, compute the upsert driver table, downfilter the original query, and compute the delete driver table. Care must be taken not to over-delete and reinsert. Deletes should perfectly represent only the rows that should no longer be part of the target as of query.

When coupled with the use of refresh bookmarks and the techniques described in *Tracking changes*, this approach yields an efficient and resilient pipeline. Given the intricacies involved, careful implementation and testing is essential. A unit test is critical to catch errors during development, and silent failures in production. Performance evaluation should be done step by step, with each component tested in isolation. The zero-load scenario provides a baseline benchmark for performance and correctness. This disciplined workflow ensures that the pipeline responds to change with both accuracy and speed.

Optimising Power BI load

Loading a Power BI model is often one of the longest steps. This is particularly true for models with large fact tables. New technologies such as DirectLake promise to remove this step entirely, yet Import mode remains the most reliable way to deliver a fast and responsive experience. Consequently, it is important to establish efficiency for Power BI loads.

Loading of Power BI should focus on assembling prepared fragments. It is *not* the stage to curate information. The restaurant analogy from *Meaningful fragments* makes the point. *Mise en place* is the long process before service time to customers. During *mise en place*, ingredients are washed, cut and portioned in advance. At service the kitchen completes the final assembly and plates dishes quickly, rather than peeling potatoes when the orders are made. Likewise, a Power BI load should be focusing on assembling finished fragments rather than transforming data.

There are three main ways to optimise Power BI loads:

- Using DirectQuery to avoid loading
- Efficient underlying source tables
- Partitioning for parallel refresh, with two further enhancements of rolling windows and incremental refresh

This chapter assumes the model sources data from SQL, where purpose-built views or tables act as source tables that map one-to-one with the Power BI model tables.

Using DirectQuery to avoid loading

DirectQuery for fact tables avoids the load entirely by sending DAX queries to the source at report time. To provide a responsive experience, the underlying source table often needs to be materialised as a columnstore table dedicated to this purpose. While materialising a large fact table is costly, it can also be incrementally loaded so that refresh remains minimal.

By using DirectQuery over am incrementally loaded columnstore table, this achieves the highest possible efficiency in terms of information movement. However, there are serious drawbacks for user experience:

- DAX queries become noticeably slower.
- Some complex DAX expressions are not supported at all.
- Certain DAX functions behave differently in Import mode versus DirectQuery, such as in how blanks are treated.
- Power BI limits how many rows can be retrieved in DirectQuery for certain operations, which restricts what reports can be built.

- Dimension values can appear blank in filter lists even when the table does not actually contain blanks.

DirectQuery works best when:

- The table is far longer than it is wide, so queries touch fewer attributes.
- Queries against the table are simple - limited to sums, counts, minimums, and maximums that work well against columnstore tables through their equivalent SQL query.
- Users are not expected to browse unit records and do not hit Power BI's retrieval limits.

Where dimensions are shared by both Import and DirectQuery facts, **dual mode** must be implemented. Dual mode keeps a copy of the dimension for Import while still allowing the engine to query it in DirectQuery when needed. This ensures relationships remain fast for Import facts and correct for DirectQuery facts.

Efficient underlying source tables

One of the biggest factors influencing Power BI load times is the efficiency of the underlying source tables, which are often implemented as views. A source view should be a straightforward join of ready-to-use tables that have been indexed appropriately to support high-performance execution.

The views may have simple logic such as:

- Looking up surrogate keys to replace composite primary keys
- Backfilling null values with defaults
- Adding on-demand columns, for example [Days expired since creation] using functions like GETDATE()

However, heavy transformations such as windowing or nested logic should be avoided. These belong upstream in curated layers where they can be tested and reused. String aggregation for display should also be avoided, as this can be handled with appropriate DAX measures, as explained in *Designing Measures*.

If a source table still requires elaborate logic to stitch pieces together, the issue is not SQL tuning. The solution is to reconsider whether upstream fragments have been adequately prepared. The ideal case is to create fragments that are both meaningful and effective for loading into the Power BI model.

Materialisation is sometimes necessary to ensure the source table loads rapidly. This is especially true when data needs to be rearranged to support fast retrieval of load partitions against the partition key - a subject we turn to now.

Partitions

Defining a table into partitions is the most effective way to reduce Power BI load times when fact tables grow large.

Partitions are akin to having a large box of books from multiple publishing years. Batches of books arrive at regular intervals, and the box needs to be updated with the incoming batch. There can be new books, and old books can be updated. Updating the whole box can take a long time if the number of books is huge. Splitting them into boxes by publishing year allows a divide-and-conquer approach and vastly reduces total update time. The ability to load partitions in parallel is the first benefit of partitioning.

If boxes of books that are too old become unnecessary, they do not need updating and the oldest set can be dropped. Over time, this reduces the number of boxes to update. This ability to apply a rolling window is the second possible benefit of partitions.

Finally, if the incoming books can be sorted by year, and in each batch only a fraction of the years have new books, then only those boxes need updating. It may take additional effort to track changes from the incoming books, but this effort can dramatically reduce total update time. This ability to incrementally refresh is the third benefit of partitions.

Thus, partitioning a table offers increasing levels of enhancement that can be adopted as the situation requires:

1. **Load partitions** in parallel, immediately reducing time to load
2. **Apply rolling window** to drop off older partitions that are not necessary
3. **Incrementally refresh** only the partitions that changed

Parallel partitions

A partition key is a business date or datetime column that divides the data into intervals such as daily, monthly, or yearly. Power BI can load these partitions in parallel. Depending on the degree of parallelism, this can lead to massive improvements in load time with little effort.

To make partitions effective, the source must return a single partition quickly. If the underlying view is complex, the solution is not to push SQL harder but to move complexity upstream and materialise the result with an index on the partition key. This ensures that Power BI can request “give me this month” and receive the data immediately. Continuing the analogy, the incoming batches of books must be readily accessible by publishing year. If not, it may take longer to fetch each year’s books than the time gained from divide-and-conquer. The optimal solution is to organise the incoming batches by publishing year. In SQL terms, this means storing the source data sorted by the partition key used to define the Power BI partition.

There is a trade-off between load time and query time. Splitting one big box of books into multiple boxes by publishing year increases the total space used for storage

because each box has a minimum size that can lead to wasted space. It can also take longer to fetch books because there are now more boxes to work through. The choice of interval size between publishing year, first of month, or publish date depends on the actual volume of books in the processing.

When partitioning, new developers often confuse which date to use. The partition key must be a business date rather than a change-tracking column like [Row update datetime]. Using the same column for partitioning and change detection will push rows toward the end of the range and break the logic. Following the analogy, the date should be the publishing year rather than the year the book arrived in a batch. The latter will always be in the last batch and does not correctly update older books.

Rolling windows

When partitions are in place, rolling windows provide the next level of efficiency. Power BI allows a data engineer to define the number of partitions to keep, and older partitions are dropped from the model. This keeps the model size under control and prevents refresh times from growing indefinitely.

Boundary points in the window are error-prone. Two details deserve attention. First, if the SQL source table itself rolls history, its window must be kept in sync with the Power BI table's rolling window to avoid mismatches. Second, the boundaries of fact tables must match related dimensions and other facts. This is particularly true of ID dimensions. If dimensions are not kept in sync, they will contain primary key values that no longer exist in the fact tables.

Incremental refresh

Incremental refresh adds change detection on top of partitions so only partitions with changes are refreshed. The native Power BI interface allows the user to choose a datetime column in the table for change detection. During refresh, Power BI evaluates the maximum value per partition. This is the **polling query**. The maximum value is stored against each partition as its **refresh bookmark**. If the next polling query returns a different value than the bookmark, that partition is refreshed. There is a substantial benefit to load time if only a fraction of the polling queries differs from the bookmarks, and thus only that fraction of partitions are refreshed.

[SCREENSHOT of XMLA partition and polling query]

Following the analogy, each box of books by publishing year may have a label showing the last time it was updated. This is the bookmark. When batches of books arrive, they can be scanned for each publishing year to see if the update date on any book is greater than the label on the box. This is the polling query. This is only effective if incoming batches are organised so that it is easy to find the maximum update datetime for each publishing year.

The native approach offered by Power BI can be limited if the source table is a view that joins multiple tables, and thus there is no single column representing change detection. It is also not applicable if partitions have deletes because the change detection column cannot reflect delete datetimes. When the native approach is not sufficient, the user can specify a custom polling query with arbitrary M code.

Polling tables can facilitate rapid evaluation of a polling query for complex cases. A polling table is simply a list of partition dates and the datetime the partition was last updated. During refresh, the polling query looks up this update datetime for a partition date.

Following the analogy, the incoming batches of books may have a companion spreadsheet that tracks publishing years and the last update datetime of books for that year. During updates, this spreadsheet is consulted to decide whether that year's box needs updating.

Power BI does not support custom polling queries in its interface. XMLA-based configuration is required.

If Power BI can rapidly evaluate whether a partition needs refreshing and retrieve the data for that partition quickly from the source table, this can lead to substantial improvements in load times. Supporting these requires the data engineer to prepare the data to facilitate these queries through indexing, sorting, or even constructing dedicated polling tables. This preparation itself takes time, and any base tables or polling tables should themselves be incrementally refreshed. Consequently, optimising Power BI loads means trading Power BI load times with data source load times. If not done well, implementing these artefacts can lead to even longer time than saved through defining incremental partition refreshes.

Conclusion

Loading a Power BI model is time intensive. Using DirectQuery over a columnstore table avoids the load entirely, but DirectQuery imposes serious limits on the queries that can be made. Consequently, the most common factor for a fast Power BI load is the efficiency of the underlying source tables. These should be simple assemblies of well-defined fragments that join easily and avoid complex transformations.

When fact tables become substantial, partitions can deliver a major speed boost by enabling parallel loads. This often requires rearranging data or implementing dedicated indexes to support retrieval by partition date.

When partitions are in place, rolling windows provide further benefit by dropping older partitions and keeping load times consistent. This introduces room for error and requires attention to detail at boundary points.

Incremental refresh adds another layer of efficiency by refreshing only the partitions that have changed. This relies on a polling query to detect changes. Dedicated polling table can support instant evaluation of the polling.

Whatever the technique, the foundation for a fast-loading Power BI model is set far in advance in the pipeline. It depends on the availability of meaningful fragments to avoid complex transforms, the planned placement of indexes to support partition keys, and the careful tracking of row changes to enable polling tables.

In the ideal case, information efficiency is maximised end to end:

1. Raw tables enter the pipeline incrementally, using polling tables to track changes and annotated with change-tracking artefacts.
2. Downstream tables transform incrementally with in-sync datetimes and are designed for ease of assembly.
3. Source tables for Power BI, whether for DirectQuery or for partitioned loads, are also incrementally materialised.
4. Finally, Power BI tables themselves are incrementally refreshed using partitions, supported by incrementally refreshed polling tables for fast change detection.

All this means the data engineer plans far ahead, letting Power BI's efficiency requirements shape the pipeline even as the first table is built.

Beyond techniques

The five principles of data engineering

The earlier sections outlined patterns and practices for data engineering scenarios. They focused on the treatment of information rather than technology. The technology-specific sections are on Power BI, and even then, we stayed away from syntax to focus on fundamentals.

This focus on information rather than technology is deliberate. It points data engineers to underlying principles rather than situational techniques. This last section, *Beyond techniques*, is where patterns culminate and summarised in five principles.

The problem with shallow curation

The most common, but incorrect, approach for new data engineers is *shallow curation*. The typical steps are:

1. Take the source system and map to the target environment, adjusting for names and data types.
2. Receive a list of business requirements for reporting from stakeholders.
3. Using the source data, apply transformation rules to meet stated requirements, load them into a few big tables for Power BI. A more experienced data engineer may organise these as facts and dimensions.

This mindset has many drawbacks. Firstly, it leads to garbage-in, garbage-out. Source applications are not designed for analytical intent and never has the information content to meet business needs. Requirements stated upfront are rarely adequate to be a reliable guide to address this gap. Moreover, this approach focuses on the questions at hand rather than all the questions about the business. Finally, going straight from source to reporting tables means there are no building blocks for use in different scenarios. Using the information for a different scenario, such as ML feature engineering, requires rebuilding logic that embedded in dimensional models.

In other words, shallow curation is too easily satisfied. It does not sufficiently push the data as found to meet the demands of business intent. Neither does it break down complexity in building towards advanced levels of achievement. Consequently, the difference is not between merely “good versus poor” quality, but between what is “possible versus impossible” for a team to achieve with this mindset.

The problem with shallow curation is that there are *some* outputs. After all, the data engineer has faithfully reproduced data from the source system for users to gain access. Such an engineer may even feel satisfied for doing an excellent job when reality is far from the potential value. If business users are dissatisfied when business objectives are not fully met, it is possible to blame poor data quality at source or on the unclear business rules. The danger of shallow curation is that, like all mindsets that starts by reducing standards, it settles for less, and by settling for less, creates a blind spot for the engineer who cannot see the failure to reach excellence.

Five principles

A surer basis for data engineering can be summarised in five principles:

- Instead of garbage-in-garbage-out with raw data, **add value through expressive entities**.
- Instead of building giant tables, **create meaningful fragments**.
- Instead of stopping with what works now, **anticipate errors that may occur**.
- Instead of reacting to requirements, **build momentum through guiding attention.**
- Instead of stopping at the symptoms, **diagnose the root cause**

That is, instead of finding quick fixes to what is wrong, the expert data engineer focuses on what needs to be right for business intent—now and into the future. These can be remembered through five phrases: **expressive entities, meaningful fragments, anticipate errors, build momentum, root cause**.

It is not a matter of “fast workaround” versus “slow and proper.” It is the difference between what gets tangled up in a mess versus what will not. In the long term, the five principles to fast and flexible paths to success.

The earlier sections covered three of the principles–expressiveness entities, meaningful fragments and anticipating errors. The remaining section introduces the final principles–build momentum, root cause.

The section ends with the chapter *Hallmarks of quality*. This essay was written years before this text and reproduced with minimal edits. The essay guided the years of work now summarised in this book.

Working with stakeholders

The most common cause of project failure is the inability of the delivery team to build momentum with stakeholders. When momentum stalls, two patterns can often be found.

The first is when the delivery team asks stakeholders for a list of "reporting requirements," but the stakeholders struggle to provide one. From their perspective, the richness of their business cannot be reduced to a simple checklist. When the team insists on detailed requirements, the process reaches a stalemate.

The second pattern appears more promising at first. Stakeholders provide a clear list of requirements upfront—sales aggregated by date or region, non-compliance by product type, even precise business definitions. The delivery team implements these, and all seems well. Yet during testing, the product does not meet its purpose. It lacks core features, breaks on edge cases, or proves too complex to use.

Both scenarios share the same flaw of reacting to requirements. Despite its obvious and repeated failure, this pattern persists. The reason for this persistence is because experts are often familiar with the intricacies of their own discipline but underestimate the challenges of others, expecting simplistic answers for everything else to be given in clear-cut form so they can proceed with their part.

The world is not so straightforward. The most difficult part of a data project is not technical complexity but in thinking deeply about the business and its relationship with the data world. Asking stakeholders to enumerate requirements is convenient for delivery teams because it shifts the responsibility for this deep thinking to stakeholders, leaving developers with the easier task of implementing clear technical rules.

Instead, in a successful project, the delivery team guides stakeholders to express their intent, then works together to explore the data in light of that business intent. Through this process, the stakeholders uncover what they need to "see" from the data for them to achieve business outcomes. These discovered needs—not an initial list—are the real requirements. This shift turns a one-way process into a two-way dialogue where the delivery team plays a leading role by guiding collective attention.

The fourth principle of data engineer is therefore: instead of reacting to requirements, build momentum through guiding attention.

Guided attention

During development, the delivery team and stakeholders explore the data together to meet business intent. The delivery team contributes expertise by guiding the stakeholders in how they should "see" the data in light of business intent—hence guided attention. This dialogue is an iteration between business and data, between questions and answers, between thinking and testing, and between business and

technical expertise. The seven principles that follow provide a practical framework for making this a success:

1. Focus on trust
2. Lead by listening
3. Own the business intent
4. Anchor a vision
5. Gather around the solution
6. Design for workflows
7. Spot the 20%

Focus on trust

Many assume that the primary mode of requirements gathering is that of extracting information. In this view, the question becomes: *"How effectively and accurately can I get the information from stakeholders to build a product?"* This is the wrong focus. The first responsibility of a delivery team is to gain the stakeholder's trust.

The reasons for focus on growing trust are:

1. Trust is the context for information. Clearly articulating and refining requirements is arduous. Stakeholders will not be able to give the right information before they have first established a strong trust in the people asking for the information.
2. Relationship with others is the biggest factor for meaningful and enjoyable work. Strong relationships with stakeholders are more likely to lead to enjoyable collaboration during the development phase, and subsequently the success of the team.
3. The focus on extracting information sees the stakeholders instrumentally as a vehicle for information. The focus on growing trust sees stakeholders as human beings as an end-to-themselves with whom we work.
4. Delivery teams have recurring partnerships with stakeholders that extend beyond a particular project. The focus on building trust takes the long-term view.

Without trust, no data project can succeed. After each engagement, the question should not be *"Do we know more about the requirements?"* but *"Do our stakeholders trust us more?"*

The best ways to grow trust are transparency and active listening.

- **Transparency** means clear communication of the construction plan and regular demonstrations of progress—weekly or twice-weekly. This is akin to a home buyer seeing the building take shape.

- **Active listening** means thoughtful paraphrasing and summarising stakeholder input. When stakeholders hear their own language reflected, they feel understood and affirmed. This also happens when they are invited to check metadata, which gives them a tangible way to contribute.

A common mistake is for delivery teams to assume that the purpose of speaking is to transfer information—if they already know what stakeholders are about to say, then they can skip to the next part. This is a mistake. Beneath the surface, the stakeholder is:

1. Developing a feeling of being heard.
2. Clarifying their own thinking through articulation.
3. Growing in data literacy through dialogue with technical experts.

All these build trust. By remembering that the first responsibility of the team is to gain trust, we resist the temptation to cut people off or jump in to correct them. Patience is paramount.

Lead by listening

In some types of partner dancing, there are two designated roles: *Lead* and *Follow*. The Lead initiates all the movements and the Follow completes the sequence with an elegant response. An analogous dynamic is needed with stakeholders. The delivery team should formulate its own statements as responding gracefully to a previous statement from stakeholders. These responses typically take one of the following forms:

- **Clarifying question**: “You said X, can you please clarify whether X means…?” or “You said your team needs X—can you help us understand how this fits into your goal?”
- **Playback**: “There’s a lot in what you just said—can I paraphrase to check I’m on the right track?” or “I’ve summarised your explanation into workflows—can I share them for confirmation?”
- **Proposal**: “Based on what you said about your workflows, here are wireframes of our solution hypothesis—does this align?”
- **Amendment**: “You gave feedback on our hypothesis—here’s a refinement. What do you think?”

By responding to stakeholder initiatives, the delivery team is, in reality, leading. This is achieved by listening closely to what has been said and naturally guiding the stakeholder’s attention through a technical problem. This approach ensures:

- Stakeholders are heard, feel heard, and are seen to be heard.

- The conversation remains structured, logical, and focused on business objectives.

There will be times when the team needs to correct an error, redirect focus, or counter unconscious bias. This principle still applies. A common scenario is when stakeholders describe solutions before the problem is fully explored—for example, asking for a detailed data dump dashboard when analysis suggests a summarised view is better. Saying "Let's focus on requirements instead of jumping to the solution" sounds natural but often feels dismissive. For stakeholders, the solution *is* the requirement.

Regardless of content, stakeholders are trying to communicate something important. This should always be affirmed. Even if the content is incorrect, the underlying intent is valuable. Respond by continuing with their initiative rather than breaking off. For example:

- **Explore the intent**: "You said your goal is to detect potential non-compliance—how does the data dump help?"
- **Explore the utility**: "You want a detailed data dump—can you give an example of a workflow where this is used?"
- **Re-orient**: "Thanks for raising the data dump idea. You previously said your goal is detecting non-compliance. Can we explore scenarios where that happens, then revisit the data dump in that context?"

These approaches respect intent and apply the first principle: focus on trust. It also applies the fifth principle: gather around the solution.

This principle should never be used as a shield for blame ("I just did what you told me to do").

Own the business intent

The task of data engineering is to align data to business intent. It is an experimental process that demands the data engineer see the stakeholder's perspective "first-hand." Without this, the team will never truly see what the stakeholder sees, and the potential of the solution will remain unrealised. This act of viewing the data through the stakeholder's lens is **owning the business intent**.

The data engineer who owns the business intent must be committed to in-depth business analysis.

This document is not a guide to business analysis, except to note that business processes often conform to recurring patterns. Mastering these patterns helps the team achieve high-quality analysis quickly, even for complex scenarios. The team can do this by consistently asking stakeholders five key questions:

- **Intent**: What is the business intent?
- **Measure**: How does the business measure the achievement of this intent, either directly or indirectly?
- **Sensor**: What instruments does the business have for knowing whether things are going well or not?
- **Controls**: What levers does the business have to influence this measure?
- **Drivers**: What external events outside of does the business's control may influence this measure?

Business analysis needs to permeate the project from start to finish. For example, concepts such as "good vs bad" entities or milestones for measuring processes are rarely defined in source systems. It is unrealistic to expect stakeholders to define these upfront before seeing the data. Instead, these definitions evolve through exploration and require input by technical expertise and experience with similar problems. This, once again, demands that the team see the business problem first-hand.

Owning the business intent is an extension of the first principle of trust. Nothing gains trust with stakeholders as quickly as a team who can fluently speak the details of a business language. From this perspective, business analysis is a delivery team's way of active listening.

Whether a team owns the business intent will define its passion, drive, business knowledge, and creativity. These qualities will determine whether its output is a mediocre reflection of the current state or a solution that pushes the business forward to excellence.

Anchor the vision

The natural extension of owning the business intent is a vision for the business. A vision is a view of the value the team aspires to achieve. It is both ambitious and concrete.

Whilst having a vision is important for any project, it is particularly critical in exploratory data projects for the following reasons:

- **Improves the probability of arrival.** Complex data projects are often ambiguous and uncertain. Teams that articulate and revisit a vision are more likely to reach the intended outcome.
- **Sustains engagement.** Long projects involve hard work, such as resolving edge cases. A clear vision inspires a sense of meaning to maintain engagement.
- **Keeps debates in perspective.** Under constraints, debates are inevitable and can become flashpoints of tension. A vision keeps these debates in perspective. Teams often relax around specific problems when they see those problems as only one part of the whole.

- **Structures discussion**. Data projects are open-ended and can easily drift. Anchoring the project in a vision, and deriving each step from that vision, keeps the team on track.

Using the vision to structure discussions begins with agreement on the overall intent of the business. The project objective is situated within the business objective, which is itself a part of the broader organisation's objectives. The discussion on each delivery feature should be anchored to this hierarchy, with constant reference back to the higher-level intent. When the conversation becomes lost, the team moves up one level and reorients.

Example dialogue:
Team & stakeholder: *Introductions*
Stakeholder: *Jumps into details about needing a dashboard*
Team: "Can you start by telling us the purpose of your business, your role, and what success means to you?"
Stakeholder: *Provides some context, still detailed*
Team: "So your organisation's goal is to improve customer experience, and your team supports this by monitoring service quality?"
Stakeholder: "Yes, but also…"
Team: "Okay, so you track service quality, but you also need to identify recurring issues to prevent them. Is that right?"
Stakeholder: "Yes…"
Team: "At the start you mentioned a dashboard. What is its purpose in this context? Is it for your team's internal use, or to share insights with other areas?"

This pattern—bringing stakeholders up to the highest level and then unpacking details step by step—requires fast thinking and familiarity with the organisation's goals. It is harder than it appears, but essential for clarity.

When priorities conflict, reference to the higher-level intent provides common ground. Agreement is easier at higher levels and can be used to resolve issues. This approach works only when the conversation has been structured from the vision downward.

Unlike a building project, which relies on well-defined specifications of the target building and strict timelines to track progress, data projects are exploratory and open-ended. In this setting, a clear vision, rather than detailed specifications, plays the role of guiding the team. It serves as a north star and compass that sustains quality and momentum across a long project. For this reason, team leaders should insist on vision, quality, and momentum rather than rigid deadlines.

Gather around the solution

Data projects often waste time through too much talking and not enough doing. Instead, the best way to collaborate is to gather around a solution—start with a pen-and-paper solution hypothesis and refine it through discussion and experimentation.

The project should develop hypotheses as early as possible, even by the second engagement. Early development matters because:

- A convincing solution is the only proof the team has understood the requirements.
- Hypotheses provide a concrete point for clarifying stakeholder thinking.
- Refinement creates excitement as all parties see an early vision come to life.

A solution hypothesis is the most important part of requirements gathering because it is the only real proof the team has done the work. In an analogy of commissioning a building, it is futile if the builder only has a list of requirements (four rooms, lots of sunlight). The builder wins the contract by providing an architectural sketch—nothing less will do. No buyer would pay a mortgage deposit for a building without a sketch. Why expect stakeholders to work with a team without showing a solution hypothesis?

When projects meander, it is often because direction is unclear. The best advice for such situations: “When the project is stuck, draw a picture of the solution.”

The best way to refine the solution hypothesis is through an open workbench format. In an open workbench, the delivery team and stakeholders meet twice a week to explore the data model, experiment with new features, test definitions, and provide feedback for the next iteration. Stakeholders may come from different business areas, making the data model a centre for converging perspectives.

This format requires developers to be confident hosting discussions and responding to impromptu questions from stakeholders. The entire team should support them in doing so. When done well, the open workbench has a transformative effect: it builds trust through transparency and enables genuine dialogue.

Teams sometimes say, “Let’s not jump into solution mode.” This statement is not acceptable with stakeholders. For them, the solution *is* the requirement. When asked to describe requirements, most stakeholders are painting a mental picture of the solution. Even if it is a bad solution, it still describes what they imagine they are working toward. The team should playback the intent or seek clarification.

The error is to draw a hard line between requirements and solution. This is a distinction developers make, not stakeholders. The real distinction is between the **why** (intent) and the **what** (solution). Stakeholders will often mix both. It is the team’s job to discern one from the other.

This cannot be stressed enough: to stakeholders, the solution is the requirement—not something separate from it. In a data project, it is never too early to gather around the solution.

Gathering around the solution is the ultimate way the delivery team guides stakeholders toward clarity. It helps them see how reshaped data interacts with business intent, turning abstract requirements into something tangible. Through iterative refinement—testing, visualising, and adjusting—the solution evolves until it displays fidelity with business reality. In doing so, attention is guided deliberately, ensuring that every discussion converges on what matters most: a solution that stakeholders can recognise, trust, and act upon.

Design for workflows

It is common for data projects to deliver reports that are rarely used. Stakeholders may specify many requirements— "I want to see count of X per country"—and show excitement during development and testing. Yet after deployment, usage often drops quickly.

This happens when solutions are not anchored in real workflows. Anchoring a solution with workflows ensure that it will have usage when deployed.

Every workflow has at least two elements: an **intent** and a **trigger**. As an example of a business working at the helpdesk, see Figure 1.

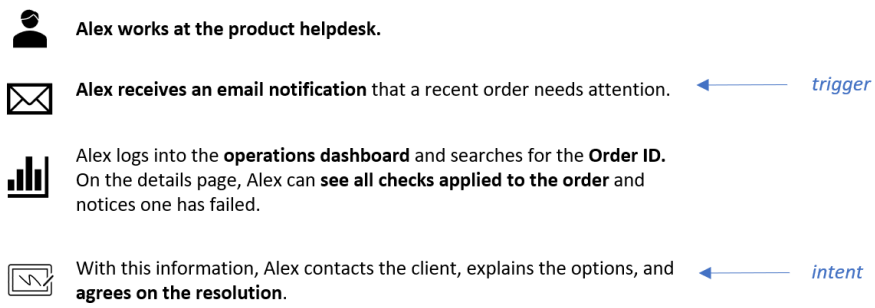


Figure 1. Example of a workflow.

A trigger can be a specific event (e.g., receiving an email from a customer) or a schedule (e.g., publishing a quarterly report).

Contextualising requirements in workflows offers key advantages:

1. Ensures the product delivers value because it meets the intent of an established workflow.

2. Ensures the product will be used because real-world triggers exist for its use.
3. Integrates the product seamlessly into the user's daily role.
4. Helps stakeholders recognise when a requirement is not important and can be dropped.
5. Provides a natural way to rationalise requirements—for example, one report page per workflow.

Designing for workflows is a simple tool to ensure that project teams are grounded in real world problems.

Spot the 20%

The 80/20 rule says 80% of results come from 20% of effort. Its flip side is that the remaining 20% consumes 80% of the effort. This creates an illusion that most work is done when the hardest part remains—the hidden, complex 20%.

The 20/80 trap is dangerous because:

1. Foundational elements are missed until production.
2. Expectations are mismanaged when stakeholders think the work is complete.
3. Technical debt piles up when essentials are discovered late.

Teams avoid the trap by:

1. Grounding requirements in thorough business analysis to uncover hidden assumptions.
2. Focusing equally on what works and what doesn't—the parts that work are often forgotten.
3. Asking not only "What should this do?" but "What could go wrong?"
4. Balancing common cases and edge cases without bias.
5. Developing in sound engineering order, not simply in the order stakeholders want to see.

As an analogy, home buyers focus on visible features—"How many rooms?" or "Is there good ventilation?"—based on obvious needs and past pain. A good builder starts with the foundation. Likewise, a good data team begins with the start of the business process and systematically work through the business processes in order, rather than jumping to flashy features.

Because of this, the team will often run counter to stakeholder instincts. This is healthy. In effect, the team acts as:

- A counterweight to natural biases that overlook fundamentals.
- A safeguard against missed edge cases that matter.

Foreseeing hidden elements is hard. It takes strong business knowledge, technical experience, and high engagement skills to counter bias. Only the best teams do it well.

Conclusion

The hardest part of any data project is thinking deeply about the business and how it relates to data. This can only be achieved when stakeholders and the delivery team explore the data together—thinking together through a dialogue focused on the solution. In this dialogue, the relationship is not symmetrical. Stakeholders, often new to data, do not know what to look for. This is why the experienced delivery team plays an important role in guiding the dialogue's attention to relevant observations.

The seven principles of engagement capture hard-earned wisdom for facilitating these dialogues. They run counter to ingrained habits that focus on visible details while ignoring complexity. They are treasures for any delivery team.

The principles change the team's view of engagement from receiving requirements and getting product sign-off to that of building trust with stakeholders. Trust is the guiding principle. Fluently speaking the business language establishes trust early in the project, gathering the solution and continuing to demonstrate progress on the business vision carries trust to the end.

These principles move the team beyond reacting to requirements. Applied well, they enable the team to guide stakeholders through complexity and build momentum toward a solution that truly meets business intent.

Construction planning

The fourth principle of data engineering is to build momentum through guiding attention rather than reacting to requirements. This can be challenging in projects that are lengthy and overwhelmingly complex. In such cases, the engineer often feels at a loss for where to begin. Because the data engineer sits on the critical path of data projects, this uncertainty directly impacts project momentum. Conversely, a data engineer who delivers outputs in an orderly and predictable way propels the project forward—participants align on the goal and instinctively sense that the team is taking concrete steps toward achieving it.

In complex projects, a data engineer stays organised and guides the delivery team by formulating a plan. This plan is expected from an experienced engineer because:

1. **No project waits in silence**. The engineer cannot disappear for a month to apply ideal patterns and return later. Unless the engineer provides the team and sponsors with confidence about the weeks or months ahead, there will be no engineering work at all.
2. **Logical sequencing matters in a build.** Doing tasks in a sensible order helps the engineer move forward with confidence, avoids code entanglement, and reduces mistakes.
3. **Only the engineer knows the effort required.** The data engineer alone understands how long quality work takes and has the sole responsibility to advocate for that time.

If the engineer does not actively formulate a plan, there will be no plan—leading the project to meander. Or someone else will create the plan for the engineer—leading the project to rush. In either case, the project suffers.

Projects can succeed or fail depending on the data engineer's plan. It is one of the data engineer's leadership roles when guiding stakeholders. Formulating a plan for the team is one of the most advanced skills to master and a mark of a confident engineer.

An effective plan

A construction plan lays out, in sequence, the components the data engineer will build and provides a forecast for how long each part will take. This plan gives users a clear expectation of what features will arrive and when. In addition, a construction plan includes non-feature components such as unit tests, technical debt clean-up, and performance tuning. The plan defines the project's momentum.

A plan is effective if it provides the project with confidence and momentum. For this purpose, the focus is on sequence rather than precise timings. While it is important not to blow out project timelines, the sequence is more critical than exact estimates

because delivering one feature after another is strongly associated with user satisfaction and project momentum.

An effective plan is akin to a travel itinerary for exploring a new country. An itinerary is not effective merely by meeting a predefined schedule; it is effective when it moves forward logically, providing a quality experience without wasting time.

A data engineer can craft an effective plan by aiming for three characteristics:

- Ambitious yet grounded
- Orderly yet flexible
- Rotating new features with backend consolidation

Being ambitious means having a view to tackle the full business processes and designing a model that addresses all questions, rather than limiting the scope to specific requirements raised by current stakeholders. Being grounded means supporting this vision with well-informed business analysis so that the plan is neither vague nor impossible, and ensuring that necessary information is available for reliable time estimates.

An orderly yet flexible plan is akin to a travel itinerary that is physically sensible (not looping back across half the country) but still organised into reasonable segments that can be shifted. For a data engineer, being orderly means isolating releases into promotes code clarity, avoid code entanglement, and minimises rework. Flexibility means structuring the plan so that key business priorities can shift forward or backward as a block without massively disrupting the schedule. This requires grouping deliverables into logical blocks centred on core business processes, rather than maintaining a long miscellaneous list of business questions.

New features excite stakeholders and propel project momentum. However, the data engineer must balance these visible features with invisible but equally important consolidation work. An effective plan achieves both by alternating between them. This rotation not only supports quality work by building consolidation into the schedule but also gives stakeholders time to absorb one feature before moving on to the next. This immersion is necessary because data insights are exploratory in nature. Testing a new feature requires users to interact with the output, grasp its implications, and check edge cases thoroughly.

Formulating an effective plan

A data engineer can craft an effective plan in four stages. Each stage involves increasing commitment to specific outcomes. A simple project may require only the first stage, while a complex project may demand a detailed plan at the fourth stage. The four stages are:

1. Discovery
2. Vision
3. Scope
4. Build

Stage 1 – Discovery

Formulating a plan begins with discovering facts about the business. The purpose of discovery is to answer key questions. At the highest level, these include:

- What are the major business processes in the end-to-end business lifecycle?
- What information is captured by each process, and where is it stored?
- What is the stakeholders' interest in each process?
- What reporting artefacts already exist, and how are they currently used in workflows?

The team answers these questions by reviewing artefacts relevant to the business processes and interviewing stakeholders.

To sharpen the focus on business objectives, the team should also answer:

- **Intent**: What is the business intent?
- **Measure**: How does the business measure the achievement of this intent, either directly or indirectly?
- **Sensor**: What instruments does the business have for knowing whether things are going well or not?
- **Controls**: What levers does the business have to influence this measure?
- **Drivers**: What external events outside of does the business's control may influence this measure?

The outputs of discovery should be expressed as **linear process diagrams** and **cumulative information diagrams**, as explained in the chapter *Anticipating Questions*.

Recall that a linear process diagram is a simple diagram that lays out the major business processes in chronological order, without loops or branches. Its purpose is to give a clear, high-level view of the end-to-end flow, making it easy to see where each process fits in the overall lifecycle. A cumulative information diagram is a table that lists business information as rows and business processes as columns, showing which process captures or inherits each piece of information. This diagram helps identify what data is available at each stage and ensures completeness for answering business questions.

Stage 2 – Vision

The vision for all data engineering projects is the same: advance business intent by obtaining insights across all underlying business processes. This vision is adapted to each project using findings from the discovery phase, especially the linear process diagrams.

For example, if the core business processes for an organisation are manufacturing, quality control, sales, shipping, and customer feedback, the vision statement could be:

"Understand the factors that drive sales profit and customer satisfaction through an integrated data source of complete business processes—from manufacturing, quality control, sales, shipping, to customer feedback. Obtaining timely and accurate insights into measures such as manufacturing turnaround, early detection of quality issues, changes in sales trend, efficiencies in shipping and latest customer sentiment."

This vision statement is ambitious yet grounded. It resonates with business stakeholders, inspires confidence, and establishes trust by demonstrating that the project team understands the business. Being neither too vague nor too specific, the vision frames the project at the right level for prioritisation and dialogue.

Such vision statements are easy to craft and extend naturally from the discovery questions. Yet many teams produce poor alternatives that lead projects to failure. For example, it is common to see vision statements such as:

"Reduce pain points of manual processes and create near real-time dashboard of the operation."

These statements are not a deep analysis of business intent but a reaction to user complaints. As such, they lack the substance to serve as a foundation for the project.

As part of vision setting, it is important to sketch pen-and-paper wireframes of sample reports that user can build off the data. The wireframe is a play-back what the team has heard, and the beginning of a solution hypothesis. Its purpose is to build trust that the team has understood the needs and inspire the project to strive for the finish line. It is akin to an artistic model of a public building prior to build—a good model generates excitement and builds public confidence. To achieve this purpose, the wireframe itself needs to be ambitious yet grounded.

When the vision is framed as above, defining the project scope becomes straightforward.

Stage 3 – Scope

The vision is an ambition to cover the business processes end to end. It is a visionary statement of the linear process diagram.

The full vision may be too large to tackle in one project. The scope defines a subset of business processes to focus on. For example, rather than covering manufacturing through to customer feedback, the project scope may concentrate on the first two—manufacturing and quality control—or on priorities such as sales and shipping. Dividing the vision this way allows the creation of project phases: the first phase focuses on two business processes, the second on the next two, and so on.

The selection of business processes should consider three factors:

1. **Availability of source data.** If the information captured by a business process resides in systems that are difficult to access, it may need to be deferred to later phases.
2. **Order of the business lifecycle.** Where possible, work in the natural sequence—manufacturing, then quality control, then sales, and so forth. Information accumulates along the lifecycle, and working in the same order creates building blocks for the next stage.
3. **Business priorities.** These often pull toward the tail end of the lifecycle, where most activity occurs. The data engineer must judge based on dependencies. For example, starting with sales without manufacturing may be acceptable, but starting with shipping without sales orders is not—no matter how much users want insights on shipping delays.

The cumulative information diagram, which tracks the flow of information and its storage, is useful for informing scope decisions.

The scope must include a time estimate for project completion. This requires input from an experienced engineer who can make a judgment based on discovery findings.

Like the vision statement, the scope is a straightforward application of the linear process diagram.

A common error is to scope by system. For example, rather than working through the business process one by one, the team works through integrating systems one by one. This is the wrong perspective. One reason, amongst many, is that it easily leads to a situation where the project creates an unusable product because the system does not have the minimal set of information to describe one business process.

Stage 4 – Build

The final stage is the construction plan. This is the practical roadmap for delivery that translates scope into actionable steps and sets out how the project will progress.

A construction plan is akin to a six-month travel itinerary in a large country. The country is the organisation, and the towns are the major business processes. Features are the specific sites within a town. Building blocks are times of being en route, such as taking a long bus, while technical consolidation is the time of rest in the journey.

This travel itinerary is exploratory in nature, so it needs flexibility, but it also needs order so that the physical route makes sense and there is a continuous flow of experiences alternating with rest. This cannot be achieved if the itinerary is too vague: “Let’s see the country.” Nor can it be too detailed: “Here’s a long, ad hoc list of things we want to do all across the country in the next six months.” Instead, an orderly yet flexible itinerary

sketches out the major towns and important sites, while the details of visiting a town only need to be fleshed out when getting close.

If the itinerary is well organised, it can be changed mid-way, such as swapping the order of towns or sites, with minimal disruption to the overall journey.

An engineer guiding stakeholders through a data project is akin to a tour guide crafting a travel itinerary for a traveller experiencing a country. The analogous concepts apply.

The two primary considerations are: **how the delivery is divided into releases, and the sequencing of those releases.**

At the highest level, build should proceed in the sequence of business processes. For example, if the scope includes manufacturing, quality control, sales, and shipping, then the build releases should follow this order, process by process. This contrasts with tackling a mix of manufacturing and sales, or sales and shipping, in one release. If the data engineer finds this cannot be done, then the abstraction of business processes may need to be revisited. For example, if the data for sales and shipping are so entangled that they cannot be separated for build, then it is arguable that sales and shipping in the organisation are better seen as one process.

If possible, proceed in the same order as the business process. For example, avoid jumping to shipping and then doubling back to sales. However, if reverse order is necessary due to business priorities, then organising the build along the boundary of business processes can still avoid creating a mess.

The releases should be further broken down within a business process. For example, a complex project may expect to build 50 tables in total; this may be divided into releases of 3 to 8 tables at a time. The more challenging the project, the smaller the release size. Each release should be centred on a group of business information to expose. For example, if the scope is manufacturing, one release may focus on basic attributes such as product types, production status, and production date; another release on manufacturing details such as material inputs and outputs; and another release on aggregated information such as time to manufacture. In grouping releases, priorities should be given to being as logical as possible from a business point of view.

Once defined, the construction plan must build in order of the pipeline by following table dependencies. This means a gradual increase in computation complexity. For example, the first release focuses on basic attributes with little transformation, the second release adds more complex computation, and a third release aggregates information from prior tables. This is the natural progression for a data engineer following the filter -> compute -> reduce steps explained in *Creating information*. This gradual layering of computation keeps complexity manageable and ensures each part is tested systematically along the way.

In a complex project with many attributes, it is bad practice to work on all the underlying tables and then all the Power BI tables and measures as one release. Instead, Power BI releases should be interleaved as information is curated in the pipeline. This means that features are surfaced to users continuously, so they get exposure to business attributes as soon as they are added to the pipeline.

Releases should be tightly coupled with unit tests, fault tolerance, and metadata. Whenever a feature is released, unit tests and metadata should come with the release, rather than as a bulk afterthought.

Releases should be kept short in time—approximately two weeks, or three weeks at most. They need to be small enough for thorough code review.

In a construction plan, not all releases will be user features. There is also important backend work such as building blocks or performance tuning. It is important to earmark the feature releases because they are the key driver of project momentum. Users should know when features are coming and receive them continuously in small chunks. User features are typically exposed to users in the self-service Power BI model.

User features should rotate with backend work. Not only does this keep up project momentum, it allows data engineers to spend time on quality consolidation and gives users space to fully immerse in testing features between releases.

By anchoring the construction plan on the user features, and keeping in mind how releases build up into a user feature, it becomes easier to reshuffle development as priorities change without ending up in a tangled mess.

Finally, it is not necessary to forecast the plan in full detail for the entire project. This rigidity is not fit for the exploratory nature of data analytics. At any given moment, only the next four or so releases need to be planned in detail, while features further out do not need detailed planning. For example, when working on manufacturing, the data engineer should know the plan for the next few releases to complete the work on manufacturing, while the further processes such as sales and shipping only need to be noted.

A construction plan designed in this manner is, like a good travel itinerary, both orderly and flexible.

The following is an example of a construction plan for manufacture and quality control.

Release	Information focus	Description	Artefacts

1	Manufacture core tables	Basic tables from source system to describe the manufacture process, with metadata and minor clean-up	Cake.Manufacture Cake.RefManufactureSite Cake.RefManufactureStatus Cake.ManufactureStep Cake.RefProductType
2.	Manufacture SLA tables	Define milestone points in the manufacture process, and compute time between milestones and total-time-to-manufacture by aggregating over Cake.ManufactureStep, compare the results against SLA for different product types	Cake.ManufactureBatchMilestone Cake.RefManufactureProductSla
3.	Quality control core tables	Basic tables from source system to describe the quality control, with metadata and minor clean-up	Cake.SampleTesting Cake.RefQualityControlCriteria Cake.RefQualityControlResult
4.	Quality control summary	Aggregate results from different quality control criteria testing, to compute an overall result for a batch, with the concepts of failure severity	Cake.ManufactureBatchOutcome Cake.RefBatchOutcome
5.	Power BI model for exposure	Create the views that define the fact and dimension tables for a Power BI report, and expose them in a Power BI model with corresponding measures, including metatada	Driver views for facts & dimensions: PBI_VIEW.Manufacture PBI_VIEW.QualityControl PBI_VIEW.ReportingCalendar PBI_VIEW.BatchOutcome PBI_VIEW.ProductType etc Corresponding Power BI tables: Manufacture, Quality control, Reporting calendar, Product type etc Measures

Stakeholder requirements

A typical IT project is driven by stakeholder requirements. But stakeholder requirements cannot determine the whole plan, any more than a town planner can plan a town-build simply by collecting a list of requirements from citizens. On the contrary, the town planner is expected to contribute expertise in dialogue with community.

In practice, stakeholder requirements swing between two extremes—too vague to guide development, or too specific so that the forest is lost for the trees. Consequently, the delivery team should not react to the requirements and build the first thing that is

asked. Such an approach is destined to lead to a tangled build if the requirements are too specific, or meandering if they are too vague.

Instead, the data engineer takes stakeholder input as part of a broader consideration for a data model that anticipates questions across end-to-end business processes. This can be done by reframing stakeholder requirements as objectives about core business processes. If the requirements are too specific, such as a long list of details like "I want to see count of sales by product type," they can be brought up a level by categorising them as objectives on sales. If the requirements are too vague, such as "I just want to know about my business," they can be made more concrete by articulating objectives on key processes such as sales, manufacturing, or shipping.

In short, stakeholder requirements should be reframed to the level explained in the chapter *Anticipating questions*. The delivery team has responsibility in guiding the stakeholders comfortably on this journey as described in *Working with stakeholders*.

Conclusion

In traditional IT projects, data engineers are often seen as "doers" who implement specified rules. This is far from their full potential. Being closest to the data, they actively shape how the organisation understands its business by reorganising information at a fundamental level. Data engineers are true influencers.

This chapter continues the theme of a data engineer's influence. In a high-intensity project, the data engineer—working with other team members—formulates a plan that not only addresses business requirements but reshapes them, using this plan to build project momentum. The data engineer is doing the build, and only the data engineer can craft the plan.

The way to create an effective plan is to use the tools explained in *Anticipating Questions*: linear process diagrams and cumulative information diagrams. These tools break down organisational complexity into a level that is useful for all parties and especially conducive to the data engineer shaping data under these frames. In this sense, the linear process diagram becomes a common ground between business stakeholders and the delivery team for planning and prioritisation.

An effective plan lays the groundwork for success. In a complex project that requires a defined scope, a rough estimate is that:

- 50% of success is determined by the scope
- 35% by the construction planning
- 15% by the actual development work

This is no surprise. As developers mature, development becomes more mechanical and predictable. Familiarity with patterns makes implementation routine. The more experience builds, the more developers can foresee outcomes and issues in advance.

Consequently, success hinges increasingly on planning. This is why the mark of an experienced data engineer is an effective plan.

Sound judgement

Sound judgement is the defining capability of a mature data engineer.

It is tempting to think of data engineering as a sequence of technical tasks: building pipelines, fixing errors, optimising performance, and integrating systems. In practice, these tasks are secondary. What distinguishes effective engineers is not how quickly they act, but **where they act**.

Sound judgement is the ability to recognise *at what level a problem should be addressed* — whether a situation calls for surface correction, structural repair, or a reconsideration of what "good" even means. It is exercised continuously, not only when something is visibly broken.

For this reason, troubleshooting is not a special activity reserved for failures. It is the most explicit expression of a posture that pervades the entire discipline. The same judgement used to diagnose a broken pipeline is used when deciding how to model an entity, where to aggregate information, or whether a system should be conformed at all.

This chapter presents a simple diagnostic framework that makes sound judgement explicit. This framework defines the final principle of data engineering—instead of treating symptoms, diagnose root cause.

A diagnostic framework for sound judgement

Sound judgement can be practised deliberately by distinguishing four stages that are often confused:

Stage	Diagnostic question
Symptom	What is observably wrong?
Trigger	What event caused the symptom to surface?
Root cause	What structural violation allows this to recur?
Final effect	What outcome must hold for the system to be genuinely fixed?

This framework does not prescribe solutions. Its purpose is to discipline attention. It prevents engineers from reacting to what is loud instead of addressing what is consequential.

A simple example

Consider someone learning piano.

- **Symptom:** the playing sounds offbeat.
- **Trigger:** the finger is lifted too late before the next note.
- **Root cause:** the learner relies on playing by ear rather than learning the mechanics of timing and articulation.

- **Final effect:** the piece is played according to the intended rhythm, not merely "close enough" to how it should sound.

Two observations matter.

First, the trigger is not the root cause. The trigger explains *how* the symptom occurs, not *why it persists*.
Second, the final effect is not the absence of mistakes. It is the presence of the intended outcome.

Sound judgement lies in acting at the level that prevents recurrence, not merely silences error.

Step 1: Symptoms — seeing clearly before acting
The first act of sound judgement is to **systematically canvas the symptoms**, rather than reacting to the first issue reported.

Stakeholders describe what they see. They do not describe the full boundary of failure. Beginning work from a single reported symptom allows the reporting channel to define the problem — and that definition is almost always incomplete.

At this stage, interpretation is a liability. The task is mechanical observation.

In medicine, uncertainty is met with vital checks. In data systems, the equivalent is to examine surface health before forming explanations.

Typical symptom canvassing includes:

- If one report is broken, are others broken?
- If one metric is wrong, are related metrics wrong?
- If a domain disappears, did the underlying rows disappear or only their classification?
- Did row counts change? Did distributions shift? Did null rates spike?
- Did upstream pipelines succeed? Did downstream consumers fail?

Jumping ahead at this stage is a failure of judgement. Engineers tend to do so for two reasons:

- pattern-matching based on past experience
- taking the reported symptom as a complete description

Both lead to work that is precise but misdirected.

Sound judgement at this stage consists in restraint. Seeing clearly comes before acting decisively.

Step 2: Triggers — locating the point of exposure

The second act of sound judgement is to identify the **trigger** — the event that caused the symptom to surface.

Triggers are time-bound and concrete. They narrow the search space and provide orientation.

Common triggers in data engineering include:

- schema changes in source systems
- new or invalid records entering a load
- code changes or refactors
- infrastructure or platform migrations
- permission or security changes
- changes in load ordering or scheduling

Triggers are useful because they are often discoverable, and because they sometimes allow temporary containment: rollbacks, bypasses, or delayed execution.

However, sound judgement requires a strict distinction:

A trigger explains when a symptom appeared.
A root cause explains why it can reappear.

Treating the trigger as the cause leads to fragile systems that fail again under different conditions.

Step 3: Root causes — acting at the level that matters

The third act of sound judgement is to diagnose the **root cause**.

Root causes explain recurrence. They are structural violations that allow different triggers to produce the same class of symptom.

Root cause diagnosis may be:

- **Theoretical:** reading the system and identifying what must be true for the failure to occur
- **Experimental:** isolating variables through controlled tests

Both require skill. Theoretical diagnosis depends on structural understanding. Experimental diagnosis depends on technical competence and disciplined inquiry.

A useful rule is:

Root causes violate core principles that define the ideal system.
Triggers merely activate those violations.

In data engineering, recurring root causes include:

- incorrect or ambiguous grain
- missing, unstable, or misused keys
- semantic mismatches hidden behind identical column names
- wide tables that collapse unrelated concepts
- implicit dependencies on ordering, timing, or existence
- weak business processes that produce unreliable identity
- compounded logic that mixes extraction, transformation, and aggregation

Two engineers may observe the same symptom and trigger. One applies a patch. The other reshapes the system. The difference is not effort, but judgement.

Sound judgement is the willingness to act where the system is wrong, not merely where it is noisy.

Step 4: Final effect — knowing when you are done
Between diagnosing root cause and completing work lies the act of applying a fix. Fixes vary: containment, refactoring, redesign, or process change.

Regardless of the fix, the final act of sound judgement is to **check the final effect**.

This includes:

- **visual validation:** not “does the code run”, but “are the outputs correct”
- **environment validation:** behaviour across development, pre-production, and production
- **end-to-end validation:** whether the information now supports the intended decision

A critical rule applies:

The final effect is not the absence of the symptom.

Error suppression creates silence, not correctness. Silence is often mistaken for success, until the failure reappears elsewhere.

Knowing the final effect is difficult because it requires seeing what “good” should be. Engineers often fail to recognise slow systems because they do not know what fast

systems look like. They fail to recognise weak models because they do not know what expressive models look like.

In data engineering, the final effect is always the same:

The business receives the information it needs to make the decision.

Pipeline success, test success, and error absence are necessary — but they are not the end.

Sound judgement throughout data engineering

Sound judgement is not exercised only when something breaks. It is present throughout the discipline of a data engineer:

- **Expressiveness:**
  - *Symptom-level response:* rename columns or add documentation when reports confuse users.
  - *Root-cause response:* reshape the model so business meaning is explicit in its structure.
- **Fragment modelling:**
  - *Symptom-level response:* patch special cases into existing tables as complexity grows.
  - *Root-cause response:* extract independent fragments and isolate responsibilities.
- **Reference data and conformance:**
  - *Symptom-level response:* reconcile mismatched numbers in reports.
  - *Root-cause response:* introduce shared reference data and apply it consistently.
- **Aggregation and grain:**
  - *Symptom-level response:* simplify visuals or add filters when reports are slow or inconsistent.
  - *Root-cause response:* aggregate correctly in the pipeline at a grain aligned with decisions.

In each case, nothing is “broken” in an obvious sense. Yet judgement is still required. Mature engineers recognise these situations early and act at the level that prevents future failure.

Conclusion

Sound judgement is the ability to act at the right level of abstraction, at the right time, for the right reason.

The diagnostic framework introduced here makes that judgement explicit:

1. Canvas the symptoms
2. Identify the trigger
3. Diagnose the root cause
4. Check the final effect

Common failures arise from:

- acting on symptoms without understanding scope
- mistaking triggers for causes
- equating silence with success

The goal is not to avoid problems. It is to address them at the level where they stop recurring. Thus, **instead of treating the symptom, diagnose the root cause.**

Sound judgement is not a talent. It is a discipline — and it is the thread that runs through all effective data engineering work.

Automation

Getting started as a data engineer

The full set of concepts and patterns can feel overwhelming for a new data engineer. It is not necessary to master everything upfront. In fact, a good data engineer is defined by much more than technical proficiency.

Starting out as a data engineer is less about mastering every technique and more about developing the habits and perspective that sustain growth. The priority is to internalise the engineering principles until they become second nature.

Developing good habits

New engineers often focus on reaching great technical feats. This is the wrong approach. Instead, all new engineers should aim for two goals—exposure and technical discipline.

The greatest and most common danger for engineers is becoming narrow—focusing purely on engineering techniques rather than the business or wider architectural perspectives. Early exposure to diverse aspects of a data project and team is a healthy antidote.

The second most common danger for engineers is focusing on flashy techniques. On the contrary, the best engineer is built on a foundation of consistent discipline in technical hygiene. That discipline is formed early through small problems.

Exposure and technical discipline can be developed through the following habits:

Aim to be helpful, especially on small tasks. A wide range of small tasks gives exposure to the scenarios an engineer deals with. The best tasks involve addressing unit-test failures or responding to low-urgency production errors. These build the mindset of anticipating errors through exposure to fixing them.

Talk to the team. Constant and open conversation with team members contributes to exposure. Engineers on the frontline are often the first to see an issue. Consequently, the ability to raise risks and articulate uncertainties develops through working with a team. Learning from experienced team members is a core part of gaining exposure, though not all team members are good models. It is important to be aware of their track record. In any case, no engineer should work silently in isolation. A new engineer is able to develop good habits of communication before the habit of solo-working takes hold.

Talk to the business. A new engineer benefits from learning the business problem "first-hand," as if part of the business team. This is owning the business intent. Instead of focusing solely on code, early habits should include writing quality metadata and descriptions. Confidence in live demonstrations and hands-on experimentation with the business grows over time and requires comfort in acknowledging ignorance or mistakes.

Understand the architecture. Developers often focus only on the development environment rather than the complete architecture. Yet multiple environments, CI/CD practices, and orchestration of loads all influence the engineer's work. Thinking a few steps ahead—not just about development but also about deployment and monitoring—becomes an important habit.

Technical hygiene. Technical hygiene begins with code management. This starts with elementary habits such as using Git version control as second nature. The most common error for new engineers is creating massive branches that release many features at once. Smaller, logically isolated releases build the capability to break down complex problems into simplified steps.

Check, don't assume. New engineers often make assumptions—about data, people, business, communication, or architecture. A habit of checking every step carefully is a prerequisite for tackling complex problems.

Estimate and refine. Engineers often shy away from estimating effort and time-to-complete, developing an unhealthy habit of making excuses for why estimation is too difficult. Yet the ability to estimate accurately is one of the best indicators of technical competency. This will not come on day one. Developers commonly underestimate before work starts and overestimate once underway. A useful habit is making a rough guess of time-to-complete before starting any task and checking the estimate afterward.

Focus on the principles. Data engineers must develop mastery of techniques necessary for quality engineering. Most new engineers learn better by doing rather than reading a book. However, understanding techniques through a theoretical framework from first principles remains important. Instead of being satisfied with ticking off a delivery feature, a new engineer benefits from reflecting on how the task links to the patterns in this book, especially the principles of data engineering.

Conclusion

A new engineer benefits from balancing the mastery of techniques with confidence in the span of responsibility. A data engineer is more than a developer applying techniques to build a data model. The role involves engaging with the business, understanding its perspective, and alerting the team to issues that may not be visible. Being helpful remains the best attitude for a new engineer.

Mastery of techniques is essential, though it develops over time. Progress comes from reflecting on these techniques consciously from first principles. This theoretical understanding forms the foundation for tackling more complex tasks.

Progress for a new engineer can be measured by the ability to estimate effort. The ultimate sign of success is the capacity to formulate an effective construction plan.

This does not require to-the-day accuracy in scheduling, but rather an orderly and flexible plan that provides clarity and builds momentum for a project team. Reasonable precision in estimates is sufficient for this purpose.

The ability to formulate an effective construction plan is built on consistent technical discipline, and that discipline begins with the earliest practices of a new engineer. These habits are not optional—they serve as the foundation for everything else in this book.

Closing essay: Hallmarks of quality

As developers, and broadly as a team with the goal of delivering products, we are under the constant temptation to build products that answer to the demands of the day. These demands are often driven by emotions we feel towards other people. They may be happiness of appreciations from a client or the fear of reprimand if we do not deliver on time.

However, the instinctive reactions of personalities, and especially of personalities who barely understand our craft, are not reliable guides to quality. While these reactions may be justified—for example, if a project has been dragging on too long, then the people around us are right to be angry, they cannot be relied upon as the quality of work.

Our work is of quality if it enables the organisation to make sound, positive decisions. Nevertheless, this is too abstract a goal for day-to-day work. Even other measures such as product usages or sentiment analysis of feedback are too distant to help developers make decisions in their daily work. We must seek other guides.

Our purpose is to lay down hallmarks of quality that help a team to recognise and celebrate quality. They are nothing new but are time-honoured practices.

As a developer, we must deliver the features, but also complete the following:

1. Expressive entities
2. Well-written explanation
3. Thoughtful unit-tests
4. Monitor assumptions
5. Anticipate errors
6. Adhere to patterns
7. Optimised code

These are the hallmarks of quality. When done well, we have not only delivered a feature but delivered quality work. The work will be high value, long lasting, will win trust and importantly, give us pride in the work we do. Not everyone will carry out all aspects in every instance, but as a team, these are non-negotiables to quality.

Expressive entities

When looking at data, it is easy to forget we are dealing with a real world. Yet the real world is what decision makers deal with. All models, whether they are simulation models or data models or machine learning models, are models *of the world*.

Correspondence to the world by the agent trying to influence it is therefore the final arbitrator of whether a data model is successful.

Expressive entities refer to the idea that the tables and relationships we create in the data warehouse should not merely reflect the data as we found it but corresponds strongly to real world business processes. It is expressive because the consumer or reviewer of the model can clearly recognise the world the model is trying to approximate. If a competent layperson cannot easily relate the model to the real world, then the model is not expressive.

The process of creating expressive entities happen at a local level when we use table and column names that are business meaningful. At the level of a single model, we often say that we are “crafting” entities or tables. At the warehouse level across several models, we often say that we are conforming entities.

A data model is not a quality data model unless it is a useful representation of the world. A data model is, therefore, a *worldview*—more than a worldview, but not less. A developer without a strong worldview cannot create a sound data model.

In a rapid development environment with the pressure to deliver, crafting and conforming entities are difficult tasks to achieve. They are hard because it is hard to think clearly about the world in a systematic way and then articulate it through data. There is a temptation to keep moving on to the next feature. Moreover, the benefits of crafting entities are long term and not easily understood by stakeholders. People understand testing and monitoring assumptions, but what does a “conceptual model” mean? However, the implication of inexpressive entities is that no one had spent time thinking about the world and the processes with which we try to influence it.

A quality data model is only achievable by a developer who has thought seriously about the world, how data relates to it, and how to organise the world in a way that makes sense for decisions. The ability to create data entities that are accurate and expressive of the world is an infallible mark of quality.

Well-written explanation

Documentation is particularly important for data analysis (as opposed to say, a game programmer) because data is a projection of real-world processes and a data code is acting as an "interpreter" between data and the world. This interpretation has a plain English equivalent. Documentation is not simply "code comments" from one developer to another but is an articulation to oneself and to others how the output is reflecting reality. The writer C.S Lewis said, “If you cannot say something in common language, you either do not believe in it or you do not understand it.” If we cannot explain the logic behind our code in plain English, why should anyone believe we know what we are doing?

The primary purpose of a well-written explanation of our code is to articulate the logic back to ourselves. What are the entities we created? What do they really *mean*? What assumptions have we made? Writing a clear explanation is to step back from the code and do the work again from a different angle. Often, in articulating an explanation, one finds new ways of approaching the problem or discover assumptions that we missed.

The secondary purpose of documentation is to win trust. To non-developers, our work is a black box. A well-written explanation is part of the transparency that are integral to trust. Business areas also have a clearer understanding of the quality work and effort put in development. These builds trust.

A developer does not need to writer documentation alone. It is the one task where other people can be intimately involved. Sharing and collaborating on documentation with subject matter experts are avenues to clarify concepts and listen to feedback.

It is interesting to see a shift of habit as developers mature. A new developer often write code with no documentation. Over time, the developer gains a habit to write documentation *after* completing code, then a habit to write documentation *together* with code. When mature, it is common for a developer to *start* with code documentation for complex scenarios (usually for the purpose for circulation). There is nothing surprising about this. The transition merely reflects the fact that the developer has mastered patterns in the work environment and is shifting focus from the mechanics of code-writing to the reality behind what the code is about.

Thoughtful unit-tests

At the minimum, writing proper unit-tests means that developer has solved the problem more than once and from different angles. This is enough to justify the writing of the test. Often in the writing of a test, the developer discovers a logical error or edge case that escaped attention during development.

In the long term, tests exist because the world is not static. What is true at the time of writing the code may not be true the day after. The data itself can change, or the code itself may change. Sometimes these changes can lead to significant errors that lead to incorrect decisions or harm trust.

In isolated instances, one code element may have low probability of going wrong due to external changes. When running several hundred thousand lines of codes against several thousand entities, and a dozen developers are changing several hundred lines per week, the probabilities of error add up.

Tests are about quality and not quantity nor coverage. Poorly written tests can lead to false confidence.

Resist the temptation to delay a test. If instinct says a code should be tested, do not delay to the next week. It is astonishing how often the test of which we think “I probably

should write this test, but I will do this a bit later" is the one that would have caught an error on the week's product release.

Assumptions are monitored

All reasoning involves assumptions. Dealing with incomplete data especially requires assumptions. Assumptions is the *X* in the "If *X*, then *Y*." Assumptions is the short-cut to *Y*. They allow one to defer insignificant problems to a future date.

The world is not static. Assumptions valid at the time of writing the code may be invalid the next day. When assumptions fail, the consequence is always negative. This is because if the failure of the assumption has no impact, then the assumption is not necessary in the first place.

The failure of assumptions is often incremental rather than sudden. The world and its data usually drift from where we thought it would be rather than making a drastic break. This is particularly relevant with the rise of AI—models that work today will behave differently as the world changes.

Whether they are sudden or gradual, we must monitor assumptions to mitigate the impact for the inevitable day when they fail. If we interpret fault-tolerance as the ability to go ahead despite errors, then assumptions monitoring is a form of macro fault-tolerance.

Developers often assume uniqueness and existence of data. Without such assumptions, there can be no analysis. But what if the product table is not unique by [Product ID] tomorrow because the application had to add a [Product sub-type]? This is quite possible if the source table came from a non-relational system. If this happens, our logic will fall apart. On the other hand, we become paralysed by risks if we constantly assume source tables can change drastically. Therefore, automated treatment of common types of assumptions should be built into any mature processes.

The key step to monitoring assumptions is to know when we are making one. More often, the issue is not that we are not monitoring the assumption, but that we are not conscious of making them. We assume the source tables will load in the daily ELT process. What if one of them does not load, or if they load but not in the order we expect? Another common type of assumption is to use hard-coded values in code.

Once assumptions have been identified, then it is a matter of being disciplined and monitoring them through an automated framework.

Anticipate errors

Tests and assumptions fall under the general category of error handling. The question which concerns a seasoned developer is not whether the code runs today, but how it may fail tomorrow. Many can write code against the *actual*, but writing code against

the *possible* is a whole new level. One can often identify a superior developer by an instinctive, second-nature, obsession about how the code may fail.

Even the most seemingly inconsequential actions can trigger an avalanche of unintended consequences. By learning from these errors and improving resilience of the process, these errors decrease. However, any large, structural change, are still likely to cause unintended effects. The first task of a developer then, is to expect errors to happen and to anticipate specific ones. A developer must assume that things will always go wrong.

The primary purpose of error handling is to limit damage and prevent catastrophe of ripple effects. If we make it a rule to expect errors, the easiest response is to limit the damage through fault-tolerant code. This is the task of preventing or silencing errors.

The second purpose of error handling is to reveal silent errors. Not all errors cause panic. In fact, the most dangerous errors are silent ones. They can be silent by intention or by accident. In either case, a human should be alerted to apply a fix, or the code should be designed to recover automatically before damage propagates. This is the task of detecting or recovering gracefully from errors.

The third purpose of error handling is for the team to grow continuously. Each error reveals a blind spot. When errors are handled properly, the team will be able to learn from the error and improve on processes. This may be implementing new automated checks or rethinking the algorithm from scratch. With each error the system becomes more mature and more resilient. This is the task of learning from errors. Learning from errors requires an open acknowledgement of failure and a willingness to share the lesson with others. This is why humility is important for developers.

Error handling is hard because the developer needs to look beyond the *now* and anticipate any changes in the world. It is also challenging because error handling requires an end-to-end understanding of the data architecture. The ability to anticipate, prevent, silence, detect, recover gracefully, and learn from error is a hallmark of genius. This should apply both to technical situation (what happens to the join if the column is null?) and to business rules (does the logic given by the business make sense in all circumstances?).

A delivery team that can continuously learn from errors is unstoppable. While error handling is challenging, a new developer can start by adopting the habit of expecting errors.

Adherence to patterns

There are slavish adherences to patterns that are counterproductive. However, patterns, whether they be coding styles, naming conventions, architecture, or the use

of Git, are an indispensable part of a strong team. There are bad teams that follow patterns, but there are no good teams that do not have them.

The primary purpose of patterns is to pass on hard-earned wisdom acquired by the team. In a complex environment, there are solutions that are difficult to arrive at, or are counter-intuitive or are invisible to the untrained eye. This wisdom has been acquired by predecessors at great cost. It is not proper for developers to remain ignorant of their value due to laziness in observation or a naive confidence in the developer's own judgement.

The second purpose of patterns is to create a consistent experience for end users. This consistent experience ranges from design language such as table names to more abstract experience such as joining a set of unfamiliar tables. This consistency of experience over a body of work is a felt quality that cannot be overstated. This hallmark of quality is achievable only by a team where every developer plays their part adhering to the whole.

The third purpose of patterns is to support thinking by filtering out mental noise. Yes, CamelCase is no better than snake_case for table names. We could have used [Datetime created] as well as [Creation datetime] or indeed [Created datetime]. In isolation, whether text files have hard tabs, or 4-space, or 3-space soft tabs matter little. On the other hand, when the same standard decision is applied across the codebase, the pattern creates an important stability. In fact, when patterns are consistently applied, anomalies stand out—even anomalies which have nothing to do with the pattern. During review of code that is not written in accordance with a usual pattern, the reviewer is forced to carefully parse and interpret every line and every word. These mental noises reduce the probability of detecting an error. It is astonishing how, after the developer fixes some code indentation, previously unseen logical errors stand-out.

Judicious use of patterns in a team will advance quality by promoting solutions that survived the test of time. Patterns also accelerate delivery by removing developer uncertainty and simplifying decision making. On the other hand, an unthinking adherence to patterns through a “copy-and-paste” mindset can lead to overengineering or stifle innovation. To strike a balance between individual innovation and accumulated wisdom, team members should see themselves as participants in a conversation of innovation in which each team member can contribute with an attitude of “inquire, not debate.”

Optimised code

Performant code carries out a task with as little server resource as possible. Elegant code promotes readability and simplicity. Taken together, we say the code has been optimised.

The primary purpose of optimising code is to support long term maintainability. In a rapid development environment, a multitude of non-performant code can easily overwhelm the server. If code is not elegant, we increase the burden of changing the code. In isolated instances the problem may be insignificant. Over several hundred thousand lines of code, the problem compounds.

A direct way to write performant code in a data warehouse is to use incremental extract logic. This is by no means easy. It can be difficult to do accurately when integrating several datasets. Incremental extract can influence overall design. For example, one may be selective about what tables to use in a single script to avoid a dependency that would ruin the potential to incrementally extract.

The secondary purpose of optimising code is to think about the problem again. It is harder, and takes more thought, to write simple code than a complicated one. Often in making code elegant, the developer sees the concept in a different light. The process of a re-think can lead to conceptual breakthroughs.

Not all good developers can write optimised code, but bad developers can never write elegant, optimised code. Optimised code is one of the harder hallmarks to achieve. We can all improve with practice.

Three aspects of quality

The hallmarks describe three aspects of data-based code: expressiveness, error handling and elegant code.

Expressiveness (creating expressive entities, well-written documentation)

Expressiveness is grounded in the fact that data is not a real-world entity in and of itself but is a projection of real-world processes onto computer databases. Such projections are often fuzzy, imperfect, and contingent. When we create expressive entities, our goal is to reconstruct a clear and *useful* picture of the world from whence the data came. Well-written documentation serves the same purpose by articulating in plain language how a data model relates to the real world. Analysts who continually take data at face-value (each row in the Task table is a task right?), or mechanically applying statistical algorithms from a textbook is divorced from the reality behind the data.

Error handling (anticipating errors, thoughtful unit-tests, monitoring assumptions)

Error handling is to code what seat belts and safety breaks are to cars or monitoring and kill-switches are to a power plant. They are not "nice to haves" but are indispensable components to a complex computational environment. Unit tests and monitoring assumptions all help to mitigate against the inevitable day when an error occurs. A developer creating code without considering what may go wrong is akin to a builder who adds a floor to a building without strengthening the foundations—it will work, for now.

Elegant code (adherence to patterns, optimised code)

Elegant code is key to sustainable growth. In a complex and fast-moving environment, we cannot afford spaghetti code nor free-for-all approaches to common problems. Performance-tuned code is obvious for the simple reason that the computational resources are finite.

Final words

These hallmarks of quality are not revolutionary. They have been around since code development began. The issue is not that developers do not know about them but that they do not do them because of lack of discipline or pressures to deliver.

There is the idea that doing work properly cost time and delay delivery. The retort is that doing work improperly cost more time. Writing good tests help capture errors earlier and reduces the time it takes to fix things up. Crafted entities vastly reduce the time to deliver complex features because the model becomes more powerful. Writing a clear explanation helps all get on-board, establishes trust and builds momentum. On the other hand, technical debt quickly comes back with a vengeance. At the level of a single development cycle, doing the work properly may increase development time by 20 to 40%. Over the course of several months, the dividends of quality work pay off and enable faster delivery than hasty development could ever achieve.

The developer is constantly under the pressures of deadlines. However, if one does not have time to do the work well, the solution is not to do the work poorly. The best way to achieve the hallmarks is simply consider these as part of, and not something in addition to, the work itself. Once we make the mental leap, it becomes more natural to build these into the daily work and to the pace of projects.

Example patterns