O'REILLY®

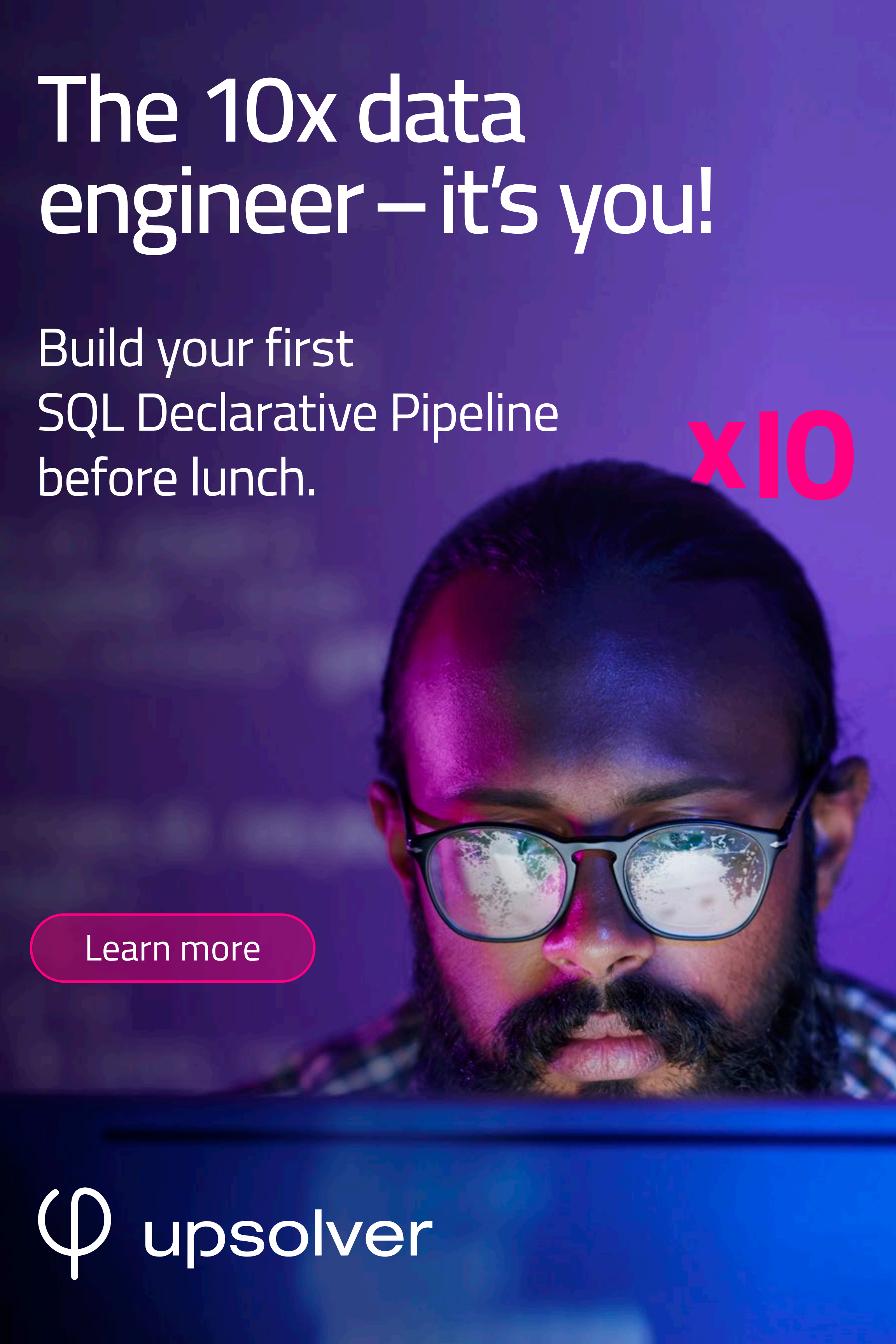
Compliments of upsolver

Unlock Complex and Streaming Data with Declarative Data Pipelines

How New Technologies Have Transformed the Way Modern Data is Processed for Analytics

Ori Rafael, Roy Hasson & Rick Bilodeau

REPORT



The 10x data engineer – it's you!

Build your first
SQL Declarative Pipeline
before lunch.

Learn more

upsolver

Unlock Complex and Streaming Data with Declarative Data Pipelines

How New Technologies Have Transformed the Way Modern Data is Processed for Analytics

Ori Rafael, Roy Hasson, and Rick Bilodeau

Beijing · Boston · Farnham · Sebastopol · Tokyo O'REILLY®

Unlock Complex and Streaming Data with Declarative Data Pipelines
by Ori Rafael, Roy Hasson, and Rick Bilodeau

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Acquisitions Editor: Andy Kwan
Development Editor: Gary O'Brien
Production Editor: Ashley Stussy
Copyeditor: nSight, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

July 2022: First Edition

Revision History for the First Edition
2022-07-08: First Release

978-1-098-13580-5

[LSI]

Table of Contents

Introduction

"Data is the new oil." Companies must be "data-driven or die." While these sayings have become clichés, they do express a truth. Our physical and digital worlds are increasingly recorded as data, and there is substantial value in converting that data into information about our business or external factors impacting our business.

However, making modern data—that is, complex and streaming data—useful has become more challenging due to rapid increases in data volume, velocity, and variety that began a decade ago. In the old days (20 years ago), the only data you could access was stored in an on-premises relational database, which was a tightly coupled storage and compute environment loaded with curated, structured data. That data could be both prepared for analysis and queried by using SQL and batch processing.

At that time, the concept of a data pipeline didn't exist. You had "ETL jobs" that moved data from various sources into a database or data warehouse while performing some light processing—cleansing, normalization, field mapping—on the data.

Today we have uncurated data, emitted in real time by digital systems and devices and stored in its raw format in a data lake. This has created amazing opportunities for analytics and technology innovation, but it also has led to skills challenges for building and managing the continuous data pipelines required to process this data into an analytics-ready state.

Unlike ETL jobs, a modern data pipeline performs continuous, complex transformations on these new types of data. What do we mean by modern data? Well, traditionally structured data originates from business transactions, accumulates through a batch process, and is held in a relational data store. Modern data is interaction data that may be semistructured, complex (e.g., nested arrays), and delivered as an event stream. Modern data is the current frontier in analytics, but building pipelines that continuously process it for analytics use has been a very complex, code-intensive endeavor.

In many ways, the wave of innovation we're experiencing in the data management industry is an attempt to recover the simplicity we had with full-stack relational databases, but in support of analyzing modern data and backed by the power of virtually unlimited storage and elastic, distributed cloud computing.

The most important innovation in this regard is the advent of declarative data pipelines, which eliminate much of the manual pipeline plumbing work (called PipelineOps) currently required for handling the scale, velocity, and complexity of modern data. A declarative approach should cut the work required to build and maintain modern data pipelines by a factor of 10!

This report addresses how we got to the current state for building data pipelines and the heavy burden it places on data engineers by requiring manual PipelineOps. It then covers new architectural patterns and technology approaches to engineering pipelines for complex and streaming data, including declarative data pipelines. It concludes with a comparison of the primary options for building modern data pipelines and criteria for deciding what to use and when in order to maximize the analytics value of your data.

CHAPTER 1

The Modern Data Landscape and Its Impact on Data Engineering

Major changes in data, data management systems, and data consumption patterns are at the heart of why engineering modern data pipelines is so challenging. In this chapter, we will cover these changes and the implications for data engineers.

Modern Data Sources

Data sources used to be limited to systems supporting business functions such as sales, marketing, manufacturing, and finance. These systems primarily recorded transactions. The data was usually sent to a central database or data warehouse each night so that reports could be run for use the next day. Besides "canned" reporting, there was also the ability to run ad hoc queries built on top of the data warehouse or data mart.

Digital transformation efforts over the past decade mean that now we can access extremely large amounts of interaction data on top of this transaction data. Interactions are events that influence whether a transaction occurs and the characteristics of the transaction. Interactions include clickstreams that record website, social media, advertising, or app interactions; logged events on digital systems such as smartphones, servers, and network equipment; and measurements from the physical world collected by Internet of Things (IoT) sensors embedded in factories and infrastructure, in buildings, or in consumer products.

A transaction is a business fact. Interaction data provides the context. In terms of size and complexity, interaction data dwarfs transaction data. A single ecommerce transaction may be influenced by dozens of web pages viewed, advertising and social media impressions, previous purchase history, interactions with a home digital assistant, visits to a store, location, the weather, etc. Besides being voluminous, these interaction data sources are often complex in structure (JSON, XML) and include nested data or arrays that cannot be analyzed efficiently in their raw form.

In terms of granularity, each interaction can be transmitted as a real-time event, upon creation, using a message queue such as RabbitMQ or Apache Kafka, or it can be added to a batch file that is processed on a periodic basis, such as hourly or nightly.

Modern Data Management Infrastructure

There have been tremendous advances in data management infrastructure technologies designed to deal with these new data sources. Here are several of the most important.

Cheap raw storage
: The ability to store data in a file or object-based system in its original (raw) format, hosted on inexpensive commodity hardware, is the basis for today's big data storage and has led to a disruptive change in data economics. Without this, modern data analytics would be restricted to only the largest and richest companies.

Elastic scale-out processing
: Being able to add or subtract processing based on real-time demand is the basis for today's more economical data processing. Cloud vendors have taken these per-unit economic gains a step further by layering on pay-as-you-go (PAYG) consumption pricing so that compute resources are utilized and paid for, like electricity.

Stream processing software
: Event and stream processing software is another innovation that has taken hold over the past decade. Apache Kafka, the best known streaming technology, is a "pub/sub" message bus that was developed at LinkedIn and converted to an open source license in 2011. It is an architecturally simple but powerful way

to input and output events onto a shared bus. Streams of events are published to Kafka topics, and these topics are subscribed to by various data consumers. Kafka operates at low-latency even while processing at high scale. Thousands of companies have now adopted open source or managed Kafka or similar technologies such as Amazon Kinesis, Azure Event Hub from Microsoft, or Google Pub/Sub.

Of course, source data has always been generated based on the occurrence of a specific event. But until recently, the event was stored as a row added to a database table that was shipped out periodically for batch processing. The limitations of traditional storage and data processing systems forced the world to batch process events.

Today, we have the technology to transmit that event upon creation as part of a real-time event stream. Lambda architecture and change data capture technologies were two early examples of event streaming being employed for practical benefit. The eventual endpoint of this evolution will be an end-to-end event-based architecture where all data sources are handled as event streams without being converted to batch data.

New Analytics Platforms and Use Cases

Systems for analyzing data have specialized beyond traditional query engines that power data warehouses. As shown in Figure 1-1, new purpose-built systems include machine learning and deep learning platforms, search platforms, and graph analytics systems. This diversity has implications for data pipeline complexity and architecture. It increases the likelihood that there will be multiple output targets for a given data source and an increasing number of change requests for a given data pipeline as new analytics uses are envisioned for the data.

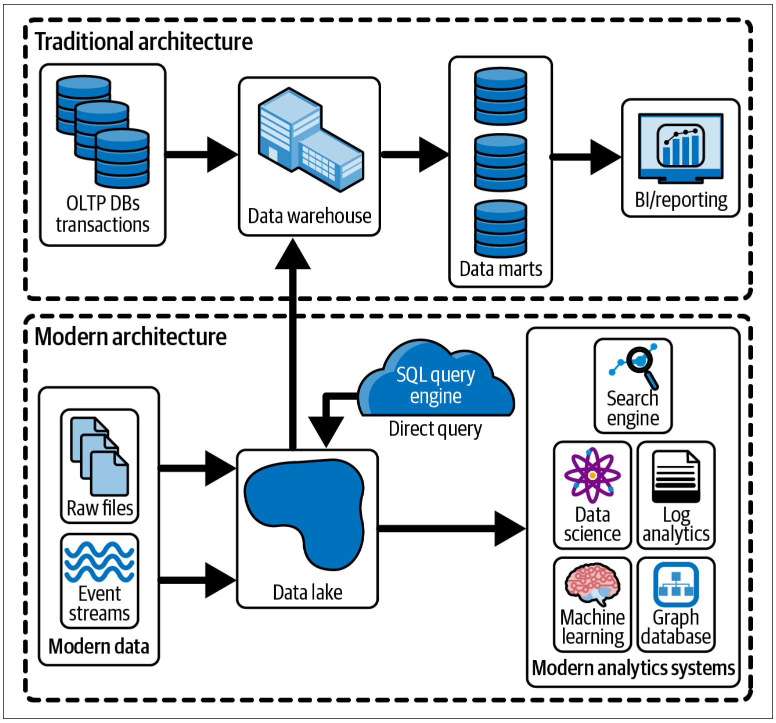


Figure 1-1. New data types and analytics use cases have forced the data architecture to evolve.

The primary use case for an online analytical processing (OLAP) database or data warehouse is business intelligence and reporting. Usually this means aggregating and filtering data to make it consumable as a table in an email report or via a digital BI visualization tool. These reports only require simple and static data preparation jobs: cleaning up the data (normalizing values, error checking); aggregating it by relevant dimensions (day, geography, product category); and presenting the data as a numeric table and/or charts.

Use cases for analytics have evolved as a result of the advances in data processing technology we've just described. Big data combined with statistical techniques allowed for data science to emerge as a way to extract insights through ad hoc analysis, data mining, and experiments. Anomaly detection emerged from real-time analysis of event streams, initially to monitor IT systems (apps, networks, services) for performance and cybersecurity threats. Real-time personalization of digital experiences is now common, where the advertisement, promotional offer, or content presented to a consumer is based on their digital history blended with third-party data on the individual. Machine learning and deep learning techniques—whereby we move from human-generated rules to machines making their own rules—are the latest techniques overlaid on top of these analytics use cases.

Data applications, which allow nontechnical businesspeople to get insights from data, are an important new use case. Data applications expose analytics to internal or external customers as a value-added service. Customer-facing data applications may not have exotic computation requirements, but their requirements for accuracy, data freshness, uptime, and response time can be quite high since they face a paying customer, who is usually less forgiving than internal data consumers.

Internal-facing data applications provide analytics to operators inside the company tasked with functions such as ensuring operational efficiency and reliability (predictive maintenance), customer service (forensics on a service quality issue), personalization of a digital experience (next best offer), GRC (governance, risk, and compliance), and risk management (fraud detection, threat detection, portfolio risk).

Decentralization of Data and Analytics

Because business needs drive data collection and consumption, data tends to pool around the operational loci of the business. This means data is predominantly sourced, stored, and analyzed locally by specific business units and divisions, not by a central group.

When the number of data sets was relatively small, it was possible to share data across domains through a centralized copy placed in a data warehouse, and then to fan out blended data sets back to business units in tailored data marts.

The centralized data lake mimics the data warehouse model, but for modern data. But as data lake usage matures, we're seeing the organizational gravity of data driving the creation of numerous domain-specific data lakes and data warehouses within a company.

Decentralization will create a need for borrowing and blending data from multiple domains. This logical use layer is one more dimension of complexity that must be accounted for when building a data pipeline architecture.

Increasing Demands for Data Freshness

It is a truism that data consumers will always demand fresher data for their analytics. In the 1980s, FedEx pioneered overnight delivery, which was initially used by lawyers and business people to expedite high-value contracts. Today, that once-novel luxury is now the standard expectation from consumers. Instant gratification is a powerful drug, and once one competitor offers speed-of-service in a given market, everyone else has to fall in line or get left behind. For delivery services, this led to a permanent shift in customer expectations as to "what is fast enough."

In the same way, data consumers press for the lowest-latency freshness they believe is available for close to free. "Overnight" delivery of data will soon have to become "up to the hour or minute." And this will drive businesses to reengineer processes to capitalize on fresher data: to act faster, become more agile, and gain a competitive edge.

Implications of Modern Data on Data Engineering

These changes in data, systems, and usage patterns create a large and growing delivery gap between the demand and supply for analytics. The growth in analytics demand is not surprising. Two decades of data management innovation—big data, streaming, machine learning, and cloud services—have spawned inventive ideas leading to new analytics projects dependent on reliable, high-performing, and cost-effective data pipelines.

Each of these projects is more complicated to build and maintain than those in the past due to the varied nature of the data, the scale of the data, the freshness requirements, and the intricacies of the various big data systems that must be employed to deliver the data.

The obvious question is "Can't we just hire and train more data engineers?" The answer is a hard "no." These massive changes in the world of data have also led to a substantial upleveling in the qualifications required to engineer a modern data pipeline, creating a huge deficit in the number of qualified engineers and a severe backlog in the delivery of analytics to the business.

The minimum qualifications for engineering modern data pipelines is rising fast. Ten years ago, a data engineer who could use SQL to query Oracle and MySQL was qualified to address most data processing needs. Today, the same data engineer needs to configure and integrate several different data platforms, write production-grade code in Python, Java, or Scala, and understand how to deploy those jobs onto distributed systems so that they perform well and affordably. Unfortunately, many data engineers aren't truly qualified to build modern data pipelines, and the ones who can are expensive to hire and hard to retain.

Not surprisingly, this gap has led to low job satisfaction. Data engineers are overworked, under relentless delivery pressure, and spend their days on detailed, manual, and error-prone programming and configuration tasks.

The negative aspects of the job don't end when the pipeline is built. They must pay down the accumulated "tech debt" of this manual labor anytime a change is required to a production pipeline, either due to a coding mistake, changes to the data (e.g., schema drift), or explicit change requests from data consumers. The workload for pipeline maintenance compounds over time since the number of total pipelines in production almost always grows. This pressure and tedium lead to burnout and exiting the field, further exacerbating the delivery gap problem.

A study published in 2021 by Wakefield Research found that 97% of the data engineering professionals surveyed reported experiencing burnout in their jobs. Also, 70% said they were likely to leave their current company for another data engineering job in the next 12 months, and 79% had considered leaving the industry entirely.

To validate that the delivery gap, we recently queried LinkedIn to compare the number of employed data engineers to data engineering job openings (see Table 1-1). If you thought data scientists were a scarce resource—and they are—you might be surprised to find that the ratio between open data engineering jobs and people employed in the field is three times higher than that for data scientists.

Table 1-1. LinkedIn Talent Solutions open data engineer jobs versus employed data engineers (February 2022)[a]

	Job openings	People employed	Job openings to employees ratio
Data engineer	28,901	69,000	42%
Data scientist	15,994	133,000	12%

[a] LinkedIn Talent Solutions query, February 2022.

PipelineOps: The Root Cause of Data Engineering Complexity

If we can't train more data engineers to build modern pipelines, can we reduce the qualifications for becoming a data engineer? Since building data pipelines is the core job for data engineers, let's unpack the skills required for that activity. Pipeline work can be divided into two categories: defining the data transformation logic, and executing PipelineOps tasks.

We define PipelineOps as the set of operations that turn business transformations into a production-grade continuous process (i.e., a data pipeline). It includes numerous sub-functions that we'll describe below. PipelineOps is the engineering required to ensure that a given data pipeline meets the established use case requirements for performance, cost, data freshness, reliability, and security.

As shown in Figure 1-2, when dealing with modern data on big data processing systems, defining the transformation logic is only 10% of the job, whereas PipelineOps is 90% of the work. This is inverse to the business value delivered by the two functions. It's as if to take an hour-long car ride, you had to spend a day tuning the engine to make sure the car didn't break down, end up in the wrong city, or run out of gas.

With all due respect to AI, defining transformation logic for a data pipeline is and will remain a human effort for all but the most trivial of use cases. Data consumers have to describe the analytics requirements for their use case and the business requirements for the pipeline that delivers on these. They will have discussions with data engineers about requirements for average and peak throughput, threshold for monitoring alerts, scaling up/down policies, error handling, isolation between workloads, recovery from crashes, and more. Having a machine make these decisions would be difficult, and this is where data engineers create tremendous value.

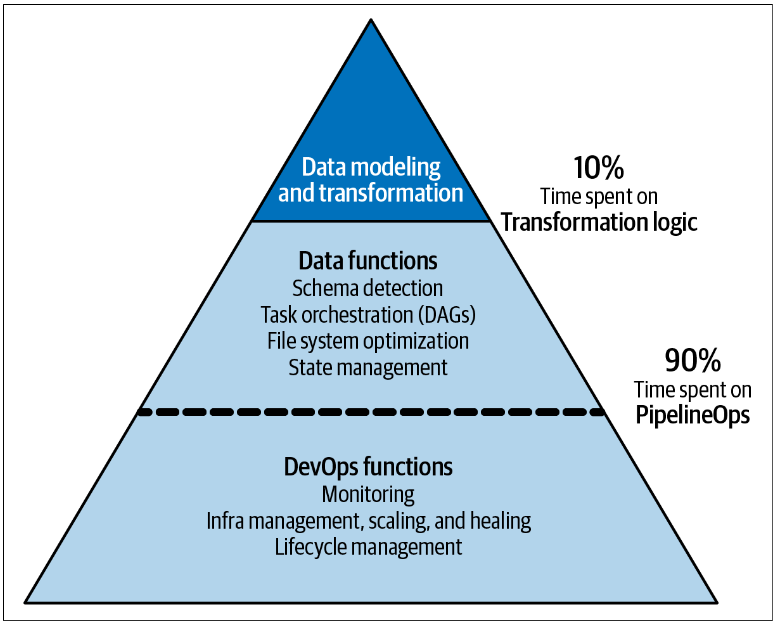


Figure 1-2. Data engineers spend 90% of their time on PipelineOps that optimize the pipeline and only 10% designing the transformations that drive value.

Conversely, PipelineOps work doesn't need to be manual engineering; that's just the current practice due to a lack of automation. In the next section, we will unpack the various functions that comprise PipelineOps, which will help clarify why it consumes the vast majority of a data engineer's time and limits modern data pipeline development to a small cadre of highly skilled big data engineers.

Automating PipelineOps is the crux of declarative data pipelines and presents a huge opportunity to improve data engineering productivity and analytics delivery. Fully automating PipelineOps constitutes a tremendous force multiplier for data engineering. It also allows data consumers to self-serve ad hoc pipelines as long as they can build the transformation logic, usually in SQL. Eliminating manual PipelineOps reduces human errors that impact data quality and pipeline reliability, thereby reducing postproduction break-fix incidents and enforcing best practices that ensure efficient, performant, and reliable data processing.

Unpacking PipelineOps

Let's dive into the work that constitutes PipelineOps. For simplicity, we split the work into data functions and DevOps functions.

Data functions

Schema detection and evolution

Data pipelines process raw source data, which is usually just a bunch (possibly millions) of files. When dealing with modern data sources, developers usually don't know a priori the schema of the data (field names, types, and relationships) or the "shape" of the data (statistics such as volume, change velocity, empty values, distribution, or first appearance in the data). Developing data pipelines without an extensive schema detection effort is equal to driving blind.

With the tools most commonly in use today, data engineers must write code and run it on samples of the data source to gain some understanding of the data set. Even then, it's only a snapshot in time.

Once in production, the concept of "schema drift" comes into play. The source schema can change without notice—a column added, a change to a field's data type, semantic changes to the data. These changes can cause a pipeline to fail or emit bad data, polluting downstream analytics. To combat this requires monitoring the source schema and manually building data validation and quality checks into the pipeline.

Task orchestration (DAGs)

Data pipelines are often executed via multiple dependent steps and staging tables, usually expressed as a direct acyclic graph (DAG). Crafting a good DAG takes time and experience. Running the execution plan as a continuous process requires constant tuning in an orchestration platform like Apache Airflow (on a data lake) or Airflow or dbt (on a data warehouse). Every change to the data pipeline, such as reacting to schema drift or addressing new requests from data consumers, forces changes across the DAG and requires substantial regression testing to ensure that the changes didn't create new unintended consequences in the pipeline.

Optimizing the object store: data lake file system and metadata management

A data lake based on object storage is the most affordable and flexible way to deal with modern data. However, unlike the monolithic database, a data lake separates storage (AWS S3, Azure Data Lake Store, Google Compute Storage), compute, and metadata (AWS Glue Catalog, Hive Metastore, Google Metadata Catalog) into distinct services. Data engineers have to spend time integrating these services, and mistakes in doing so can create inconsistencies that corrupt data.

Also, a data lake is an append-only file system, which brings benefits but also creates more data engineering. Managing a data lake to ensure consistency and high performance via low-level file system code consumes a substantial amount of a data engineer's time. A common example is properly tuning file compaction, the process of turning many small files into fewer big files that can be processed more efficiently.

State management

State management is necessary for blending multiple frequently changing sources into a single output data set. The ability to combine streaming data with big historical data is the analog to using a JOIN command in a database. The processing state can become very large in use cases with scale, yet the state also needs to be available for lookups with very low latency.

The common solution to state management for streaming data at scale is to use an external key-value store such as Aerospike, Redis, Cassandra, RocksDB, or AWS Amazon Dynamo. The data engineer is responsible for managing this additional system and the pipelines between it and the processing engine.

DevOps functions

Managing ongoing operations of data pipelines in production creates additional requirements for data engineers. This work applies to every pipeline in production, which may be hundreds for a large organization, even if only a few new pipelines are being developed at any given time. People no longer with, or outside, the company may have built these older pipelines, and the pipelines may be poorly documented or understood. In short, once in production, data pipelines are the gift that keeps on giving.

Monitoring

A data pipeline is a continuous production process that requires performance monitoring, just like an application or network. Each data pipeline requires definition and capture of metrics that are reported to the central monitoring system that the DevOps team uses to detect deteriorating performance, data loss, data quality issues, or outright pipeline failure.

Infrastructure management, scaling, and healing

Data pipelines often run on large and dynamic volumes of data, which are best served by distributed processing systems running across many compute instances. Data engineers need to manage that infrastructure, ensure healing on errors, upgrade software versions gracefully, and tune the scaling policies to optimize cost versus latency as the workload grows. As data freshness requirements move from days to hours, minutes or seconds, the pressure on data engineers to keep pipelines running reliably increases, and the margin for error evaporates.

Pipeline lifecycle management

In the normal course of operations, data pipeline logic will require updates due to expected and unexpected changes, such as fixing bugs, performance issues discovered at scale, schema evolution, or change requests from downstream users. Data engineers must implement these changes and manage the pipeline metadata to ensure version control and the ability to run a disciplined deployment process, including testing, rollout, and rollback if needed.

CHAPTER 2

Emerging Architecture Patterns

New architecture patterns have emerged to address the myriad changes in the data landscape and the gap between new requirements and old architectures. In this chapter, we first cover three foundational patterns: event sourcing, stateful streaming, and declarative data pipelines. We then discuss how these can be combined with cloud object storage to upgrade the data lake into a database for modern data. We close the chapter with a discussion of the emerging data mesh paradigm and the related concept of data as a product.

Event Sourcing for Analytics Pipelines

CRUD (create, read, update, delete) is the dominant operating model for databases, but it has two key limitations as it pertains to at-scale, high-speed data. First, there is a lack of workload isolation, meaning that writes and reads compete for the same database state resource, limiting the ability to scale the system. Second, CRUD is stateless, so there is no way to maintain or audit data lineage; you lose track of earlier versions or the changes that led to the current version. Lack of state knowledge also means that you can't "go back in time" to a previous state and reprocess data, a useful capability for backtesting a research hypothesis or retroactively changing pipeline logic to address a bug or new requirement.

Event sourcing (see Figure 2-1) is an alternative to CRUD that addresses these issues. Event sourcing is an architectural pattern in which all changes to the state of an entity are tracked by reading and committing events to an event store. The state of a database object is stored as a sequence of events starting at the beginning of the object's existence. A new event is added to the sequence each time the object changes state.

This append-only log of events becomes the "events source." Projections of that data generate "live tables" that always reflect the current state of the data. A projection is created by writing a data pipeline that applies transformation logic to the event source. Each data pipeline can run on separate resources (workloads are isolated) so performance is not limited by the number of projections created from the event source. Because the raw data is an append-only log, it is always possible to trace back the steps that generated a given projection or see what the projection would have shown at any previous point in time.

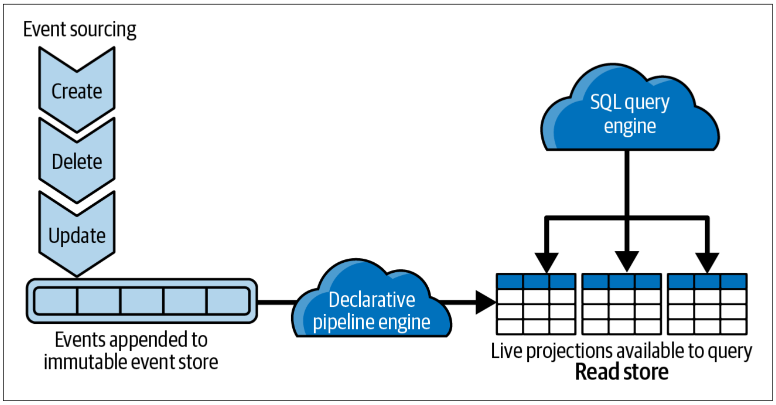

Event sourcing
Create
Delete
Update
Events appended to immutable event store
Declarative pipeline engine
SQL query engine
Live projections available to query
Read store

Figure 2-1. Event sourcing captures every state change in an entity. It allows projections of the current state and the ability to reprocess data from a previous state.

Stateful Stream Processing

The second pattern is stateful stream processing.

We are in the middle of a transition from batch-first to stream-first processing. Batch cycles are too slow for companies that want to utilize data minutes or even seconds after it was created. Increasingly, analytics use cases require blending real-time streaming events with historical data to detect changing trends and anomalies that may indicate an issue that needs immediate attention.

As streaming data was introduced to the analytics ecosystem, a pattern called a Lambda architecture emerged. It featured two parallel paths: a "slow path" for historical data stored as a batch, and a "fast path" for the most recent data delivered as an event stream. The "fast path" delivered data from the last few minutes and the "slow path" delivered the rest.

While the Lambda approach served a purpose, it also created substantial overhead because one had to maintain separate batch and stream processing pipelines. In addition to doubling the data engineering effort, it requires keeping the processing code for each path in sync as changes are made, which creates a data quality risk.

Moving beyond Lambda requires enhancing stream processing systems to deliver the same functionality as batch—namely, the ability to blend recent data with the *entire* historical data set. Usually streaming systems store only a small amount of recent data in-memory, which means older data can't be used in stream time.

To effect stateful processing on streaming data, you must implement a state store that allows you to quickly look up and retrieve the state of the data stream at any point in time. Scale is the issue, though. Spark Streaming originally stored state in-memory and on the local file system. This solution forced frequent garbage collection, which would freeze the JVM. To address this limitation, Spark 3.2 added RocksDB as a state store running on the Spark executor. This lifted the scale limitation to millions of keys, but for truly large states, Spark requires a separate key-value store. In 2018, Upsolver introduced a pipeline solution that integrated a decoupled key-value store that supports billions of keys in RAM without using local storage.

Declarative Pipelines That Automate PipelineOps

The third core innovation is declarative pipelines. In Chapter 1, we discussed PipelineOps and the data engineering effort required to turn transformations into production-grade pipelines, including activities like task orchestration, file-system interaction and management, state management, integrations to other platforms, and cluster management. These are the tasks that require expertise in big data engineering and consume the vast majority of a data engineer's time.

Recently, products have been developed that apply a declarative approach to data pipelines, using SQL as the programming language for describing the pipeline. With a declarative approach, you only express the desired outcome of the transformation but do not specify the underlying PipelineOps activities to optimize the workload. Instead, you rely on an automated processing engine to apply PipelineOps best practices for the storage/compute stack in use, usually a cloud data lake.

A declarative pipeline eliminates a great deal of PipelineOps. You only specify a source, a target, and the desired transformation outcome. Described in database language, it uses SQL for pipeline tasks in the same way one would query a source table in a database and output the result to a target table.

In addition, declarative SQL operations can be performed on streaming data as well as batch, including stream-specific operations such as window functions and streaming aggregations. Commands are also provided to support incremental updates for output tables (i.e., upserts) to keep output tables based on frequently changing data current in a compute-efficient manner.

While declarative pipelines embody a simple concept, you can see how they can revolutionize data pipelines by providing a layer of abstraction between the valuable transformation logic and the burdensome PipelineOps functions.

It is fair to ask whether you can trust an automated "black box" to perform the PipelineOps activities. Putting this in context, the "black box" that enables declarative pipelines is analogous to the "optimizer" that powers relational databases such as Oracle. This is the standard evolution of technology through automation. Low-level functions are first performed manually, and over time, best practices are discovered and then automated as code, which reduces the technical skill required to use the technology and expands adoption of the technology. In short, this is our path out of the data engineering delivery gap introduced in Chapter 1.

The Cloud Data Lake as Tomorrow's Database

Let's see how these three innovations—event sourcing, stateful stream processing, and declarative pipelines—play out on a cloud data lake. Data lakes are cost-efficient but introduce the technical challenge of optimizing a raw file system for analytics use. That's something a relational database automates for smaller, structured data sets.

The data lake (cloud object raw storage) is already the de facto standard for storing raw data. Data lake query engines deliver SQL-based query access to that data where it already resides, which has the potential to make the data warehouse redundant. The obstacle has been that unoptimized data lake tables are known to deliver slower query performance. Data lakes also limit companies to append-only models, which don't suit many use cases where the most recent value (or aggregation total) is all that is relevant.

With the incorporation of the three patterns discussed above, the cloud data lake can evolve to provide the complete set of functionality expected from a traditional database, whether transactional or analytical, while adding the ability to ingest and process real-time event data along with batch data, which is challenging for databases and data warehouses.

This new stack will be based on immutable cloud object storage used as an event-sourced entity. Projections from the event-sourced store (supported by a scalable state store) will be programmed using declarative methods, which will create mutable, strongly consistent data lake tables, ideally in an open format. These tables will act as the serving layer for transactions and analytics queries. For high-integrity transaction processing, a transaction log can be integrated into this solution to provide ACID properties.

Two other changes are necessary to create a database on a data lake. First, we need to augment the append-only model by adding an upsert capability in the projected data lake table so that it always shows the most recent values for processed outputs. Second, we have to enforce best practices for organizing cloud object storage so it can be queried at the speed of a data warehouse. That includes the use of columnar file formats like Apache Parquet or ORC, compression methods such as Snappy or Gzip, continuous compaction of small files, and proper partitioning.

These innovations often are combined with a data lake query engine under the marketing label of a "data lake house," which is often positioned as a competitor to the cloud data warehouse. But the idea is bigger than this. Because the data lake is an open architecture and supports streaming data well, it can act as a processing hub for all data (modern or legacy), with open format output tables being queried directly and/or exported to a variety of analytics systems.

The Data Mesh Paradigm

In Chapter 1 we discussed the natural tendency of data to become distributed and siloed across an organization. This has led to interest in the data mesh paradigm, an approach to data management where data is kept and prepared in the domain where it originated and is presented as "data products" that can be queried by users in the domain or other business units.

Decentralized analytics pushes the need to build data pipelines out to the business domains (as opposed to a central IT services group), where data engineering skills are generally less sophisticated (see Figure 2-2). This requires new data pipeline tools that are usable by less-technical domain users such as analysts and data scientists.

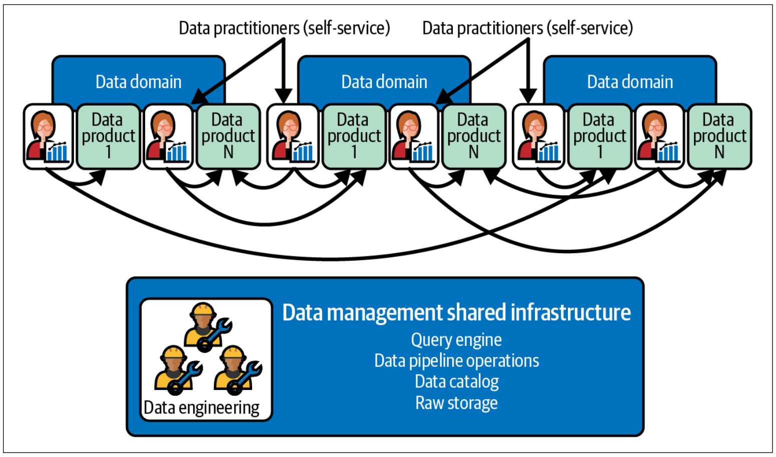


Figure 2-2. A data mesh enables decentralized data products that are shared across the organization.

A data mesh decouples the management of the data infrastructure from management of the data. It distributes ownership and accountability for data to the people who best understand it, have domain knowledge, and can ensure it is reliable and accurate. Ideally, these data practitioners have self-service access to their domain data. They build their own pipelines, they perform any necessary transformations, ideally with little to no coding, and they productize their analytical data for easy consumption.

A central data infrastructure team owns and provides the required data management systems as shared services to the domains. Each domain self-provisions these services to capture, process, store, serve, and enhance its data products.

The relevance of the data mesh to our discussion is about where control lies for each function in the analytics value chain. In a data mesh, declarative pipeline tools are of great value as they allow SQL-savvy domain users to build data products.

Data as a Product

Data as a product (DaaP) is a core tenet of the data mesh paradigm and is a separate concept worthy of a short discussion. DaaP applies software product thinking to data sets, including design-driven principles and best practices copied from the software development lifecycle (SDLC) best practices.

Productized data has the characteristics of a discrete application developed to address a specific business need. It's easy for the consumer to discover (it's cataloged), understand (the metadata of the data structure profile is exposed and stable), and use (it's optimized for popular use cases in terms of data granularity, freshness, and query performance, and is API accessible). As a "product," it conforms to cross-domain corporate rules and standards in relation to ontology/semantics, field structures, versioning, monitoring, data privacy, security, access control, and overall governance.

The adoption of a product mindset for data is also significant from an organizational point of view, as the business is establishing the role of domain data product manager and empowering the domains to be accountable for the "product quality" of the data sets they produce and share.

In short, DaaP leverages standard practices, rigor, and SLAs to deliver high-quality and accurate results that can be easily observed, consumed, and acted upon by users and applications.

CHAPTER 3

Modern Data Pipeline Alternatives

Now that we've covered some of the new patterns emerging in the world of data architecture, let's compare the different approaches available for building modern data pipelines by considering ingestion, transformation, and processing platform (data lake or data warehouse).

Criteria for Evaluating Approaches to Data Pipelines

To compare different approaches, we'll apply the following criteria.

Data source connectors
: The variety of modern data sources is a common reason for having multiple pipeline platforms. Options range from tools supporting a multitude of SaaS applications to those supporting a smaller set of general-purpose data infrastructure systems such as databases, raw file systems, and message queues.

Compute capabilities
: Batch is the traditional way to process data. Streaming usually requires a dedicated engine such as Spark Streaming or Flink. Processing data with a large state has been a limitation of stream processing, but advancements in state stores have closed the gap and allowed stream processing to replace batch in an increasing number of use cases. Transformations range from compute-light (data cleansing) to compute-heavy (joining two data streams, aggregating streaming data).

Orchestration capabilities
: Pipelines require orchestration that describes pipeline stages and dependencies as a DAG. The complexity of defining and maintaining DAGs led to the introduction of declarative data pipelines where orchestration is automatically inferred from the declared transformation logic based on best practices for the underlying processing storage/compute platform, and then implemented as code.

Storage/compute platform
: Pipeline platforms that run on data warehouses leverage the native compute engine to process data and therefore limit themselves to managing the pipeline process. Pipeline platforms that run on data lakes must provide a compute engine and a method for managing and optimizing the storage layer.

Development experience
: Except for pure data replication pipelines, all pipelines require writing transformation code. Transformations may be declaratively written in SQL or in imperative programming languages such as Scala, Java, or Python. Pipelines that aren't declarative require the data engineer to master the PipelineOps functions we described in Chapter 1, such as orchestration, file system management, state store management, and configuring distributed processing engines such as Spark.

Now that we have defined the dimensions for comparison, let's cover the common alternatives and see how they fare.

Option 1: Pure Data Replication

The simplest pipeline copies data from the source to the data store (data lake or data warehouse) with minimal transformation (e.g., normalization, cleansing). To execute major changes to the data such as aggregations or joins, a pure replication tool must be coupled with a transformation tool that operates on the data store (data lake or data warehouse).

Replication tools require reliable source connectors. Different replication tools will specialize in certain source types, such as replicating relational databases via change data capture (CDC) or replicating data from SaaS applications such as Salesforce. Providers of connector-based replication products spend much of their R&D

resources maintaining dozens or hundreds of UI-configurable connectors to keep up with periodic schema changes and API updates.

Table 3-1. Pure data replication

Data source connectors	Specialized, dozens of SaaS applications; or 6–12 relational database sources for CDC products
Compute capabilities	Basic data cleansing functions
Orchestration capabilities	Automated
Integration with the data store	Primarily cloud data warehouses Some can read and write to cloud object stores
Development experience	UI (sometimes with an API for automation)
Examples	AirByte, FiveTran, Stitch, Matillion Data Loader

Option 2: ELT over a Data Warehouse

ELT (extract, load, transform) products manage a pipeline that ingests raw source data into a data warehouse, where the data warehouse compute engine performs transformations to create analytics-ready data sets. They require separate jobs to address orchestration, data quality, monitoring, and lineage. The main advantage of ELT is that it leverages widespread expertise in SQL, enabling users with data domain expertise to write transformations using query logic.

There are three main disadvantages to the ELT approach. The first is lack of support for streaming since running queries on a data warehouse is by definition a batch process. The second is the high and variable cost of running a continuous process on a cloud data warehouse; this concern is especially relevant when data freshness is required at scale. The third disadvantage is the overhead of manual orchestration using DAGs. A pipeline is usually composed of multiple dependent steps with temporary staging data. It's not trivial to manage all of these steps together as a single pipeline. This effort is manual, error-prone, and responsible for many break-fix issues once the pipeline is in production. Schema drift in the source data and changes to downstream requirements put pressure on this architecture by forcing frequent change cycles for both the transformation and orchestration logic.

Table 3-2. ELT over a data warehouse

Data source connectors	No connectors, data ingested by a data warehouse connector or a separate tool
Compute capabilities	"Pushes down" processing to a data warehouse
Orchestration capabilities	DAG-based
Integration with the data store	Primarily cloud data warehouses Some can write to cloud object stores
Development experience	UI/Python/Jinja
Examples	dbt, Apache Airflow, Apache NiFi, Matillion ETL

Option 3: Apache Spark (Hadoop) over Data Lakes

Apache Hadoop is a framework that deconstructs the monolithic database into multiple services (storage, metadata, compute) that are managed separately and scale independently as a "data lake." Decoupling these services leads to improved economics for big data at scale and allows for open interfaces that mitigate vendor lock-in. MapReduce was the original data lake compute layer for Hadoop but has been replaced by the higher-performance Apache Spark in most cases.

The criticism of the legacy data lake approach has always been the complexity of getting the raw data to consumers in a usable form, which has required specialized data engineers who can write low-level code and optimize performance of distributed systems.

While the discussion around data lakes and data warehouses has taken on an almost religious tone, historically the distinction has been straightforward. Data warehouses offer ease of query through SQL, while data lakes offer the cost advantages of a cloud object store and less expensive compute cycles, along with the flexibility of being able to store data tables in an open format.

A data lake makes sense for use cases that aren't a natural fit for data warehouses such as streaming and semistructured data, machine learning, and high-scale continuous workloads. In these cases, the cost of data storage and data processing will probably overwhelm the added data engineering costs required for the data lake, and as discussed in the next option, the data engineering workload on a data lake can be greatly reduced with declarative pipelines.

Table 3-3. Apache Spark (Hadoop) over data lakes

Data source connectors	Infrastructure connectors—can connect to object stores, message queues, databases (JDBC), and sometimes applications
Compute capabilities	Any transformation Stateful operations at scale require a state store (key-value store)
Orchestration capabilities	Batch job orchestration requires a separate tool Stream processing is automatically orchestrated, but multiple pipeline steps create dependencies that require manual work
Integration with the data store	Data lake only Operations like compaction and updates can be employed if a standard data lake table format is used
Development experience	Java, Scala, or Python
Examples	Spark for batch Spark Structured Streaming / Flink for streaming

Option 4: Declarative Pipelines over Data Lakes

Declarative pipelines are a recent innovation offered by two vendors—Upsolver in 2018, and Databricks in 2021 through the Delta Lake storage format plus proprietary enhancements. Declarative pipelines address the complexity challenge of data lakes. They use SQL commands to define the desired transformation output. They automate orchestration, state management, file system optimization, and other PipelineOps tasks.

Declarative pipelines allow data lakes to match the "SQL simplicity" advantage that data warehouses (Option 2) have traditionally held over Spark-based data lakes (Option 3). This option was made possible due to innovations that allow stream processing to be used for workloads that previously required batch.

Table 3-4. Declarative pipelines over data lakes

Data source connectors	Infrastructure connectors—can connect to object stores, message queues, databases (JDBC), and sometimes applications
Compute capabilities	Any transformation. Stateful operations at scale require a state store (key-value store).
Orchestration capabilities	Automated
Integration with the data store	Data lake only
Development experience	SQL (declarative)
Examples	Delta Live Tables, Upsolver

Option 5: Declarative Data Lake Staging to a Data Warehouse or Other Analytics Systems

Combined use of data lakes and data warehouses is the option we expect. It combines the economic and flexibility advantages of a declarative data lake for data processing with the query breadth and performance advantages of a data warehouse. As noted above, we expect the data lake to reach parity with the data warehouse, albeit a few years down the road.

For modern data sets, many companies will store their raw data in the data lake object store, after which it can be processed into tables for performant ad hoc queries using a data lake query engine such as Presto or Trino, or exported to the data warehouse for reporting and direct query. Output tables could also be fed from a data lake to other analytics systems such as search, machine learning, or a custom data application. The processing that converts the raw data into analytics-ready tables occurs economically on the data lake, and the platform used for analytics is based on the use case requirements.

Choosing the Best Approach for You

With these categories in mind, how do we decide what works best and when? Here are four questions to ask.

First, what are your data freshness requirements? Will you need to process streaming data within minutes or seconds? If so, that will push you toward a data lake. Using a data warehouse for data preparation will be adequate if you are only processing slowly changing batch data sets where hours-to-days freshness is acceptable.

Second, what are the technical skills of your anticipated pipeline builders? Are they expert big data engineers, database administrators, or data consumers, such as data scientists or "analytics engineers"? The traditional data lake using Spark is the most taxing approach and requires the deepest experience, followed by cloud data warehouses where you have to integrate multiple tools but not manually configure a distributed system by hand. The approach requiring the least level of programming skill is an option that includes a declarative data lake where only SQL knowledge is required, with or without a data warehouse as an analytics output.

Third, which platform provides the best cost profile for your use case? Processing on a data lake using commodity compute, such as Spot instances, is much cheaper than relying on data warehouse compute. In addition, a batch process on a data warehouse often must perform multiple scans of the raw data, especially for handling late data and upserts (e.g., re-computing an aggregate frequently), while stream processing on a data lake allows these operations to be performed in stream, scanning each event only once. Lastly, since data warehouses process in batch, your processing will usually require a larger compute cluster than if you are streaming, as you have to budget your RAM for the peak batch size.

Fourth, how important are open systems to you? The trade-off is between the flexibility to take advantage of future innovations an open system offers versus the promise of higher performance today in a closed system. Data warehouses use proprietary formats to maximize query performance, but not necessarily the performance of continual data processing jobs. Regardless, your data in a warehouse is not available for use by other systems without being moved to those systems, potentially incurring a data transfer cost. A data lake using a standard format like Apache Parquet gives you maximum flexibility to move the data to other systems in the future.

Conclusion

A Declaration of Independence for Data Engineers

Data engineering has become a very popular and technically challenging field. The amount of work is growing quickly due to the emergence of modern data and the demand for analytics that takes advantage of its ability to move beyond *what happened*—the purview of transactions—to *why it happened*—the place where analysis of the interactions that drive those transactions reigns.

New patterns that provide the foundation for reliable real-time analytics—such as event sourcing and stateful stream processing—are functions databases or data warehouses weren't designed for. These add to the challenge.

Much of the manual effort required of data engineers to build modern data pipelines is a consequence of the incomplete evolution of big data systems that started with Apache Hadoop in 2006. In databases, these functions are automated "behind the scenes" through an "optimizer" function. When it comes to modern data systems, the data engineer is the "optimizer"!

The importance of declarative pipelines for big data and streaming data is that they mark a return to the model of automated optimization. This has two critical implications. First, it frees data engineers from the tedious effort of writing code, optimizing file systems, or orchestrating DAGs, allowing them to focus on more valuable data modeling and transformation design. Second, it allows anyone who knows SQL to build a modern data pipeline—including data scientists, analysts and "analytics engineers."

What does this mean for a data-driven business? Declarative pipelines will lead to a quantum leap in analytics agility as raw modern data is made available to analytics end users more quickly than in a world of hand-coded PipelineOps. Simple requests may be fulfilled the same day, and building complex pipelines may take weeks rather than months. Beyond the engineering productivity benefit this delivers, businesses will gain enormous benefit from unlocking raw data that was previously bottlenecked by the complexity of the data engineering process. In short, declarative pipelines will help companies enjoy the fruits of analyzing modern data through deeper insights and faster innovation.

About the Authors

Ori Rafael is co-founder and the CEO of Upsolver, the only no-code data lake engineering platform. He has more than 15 years of experience in databases, data integration, and big data. Before founding Upsolver, he held a variety of technology management roles in the Israeli Defense Force (IDF) elite technology intelligence unit.

Roy Hasson is Sr. Director of Product Management at Upsolver. Before joining Upsolver, Roy held a number of roles at Amazon Web Services, including Principal Product Manager for AWS Glue/AWS Lake Formation as well as roles in global business development for analytics and data lakes, and as a Technical Account Manager leading strategy and supporting implementation of data architectures with select customers. Prior to AWS, Roy spent 15 years working at tier 1 service providers to design and deploy large telecommunications and data network systems.

Rick Bilodeau is the Executive Vice President of Marketing at Upsolver. He has spent 30 years in B2B technology, with experience that spans product management, product marketing, and marketing leadership in the networking and big data industries.