ITPro Today™

NEXT

The Data Lakehouse Explained

This comprehensive five-part explainer introduces the concept of a "data lakehouse" and explores what is new and different about the model.

By Stephen Swoyer

Brought to you by informa tech

TABLE OF CONTENTS

Part 1: An Introduction to Data Lakehouses

The term "lakehouse" is a pun/portmanteau. It derives from the two foundational technologies — the data lake and the data warehouse — from which the concept of the data lakehouse itself derives.

At a high level, the data lakehouse consists of the following components:

- Data lakehouse
- Data lake
- Object storage

The data lakehouse describes a data warehouse-like service that runs against a data lake, which sits on top of an object storage service. These services are distributed in the sense that they are not consolidated into a single, monolithic application, as with a relational database. They are independent in the sense that they are loosely coupled — that is, they expose well-documented interfaces that permit them to communicate and exchange data with one another. (Loose coupling is a foundational concept in distributed software architecture and a defining characteristic of cloud services and cloud-native design. The second part of this report explores the cloud-native underpinnings of the lakehouse.)

How Does the Data Lakehouse Work?

From the top to the bottom of the data lakehouse stack, each constituent service is more specialized than the service that sits "underneath" it.

Data lakehouse: The titular data lakehouse is a highly specialized abstraction layer — basically, a semantic layer — that exposes data in the lake for operational reporting, ad hoc query, historical analysis, planning and forecasting, and other data warehousing workloads.

Data lake: The data lake is a less specialized abstraction layer that schematizes and manages the objects contained in an underlying object storage service (e.g., AWS S3, Google Cloud Platform Storage, Azure Blob, etc.) as well as schedules operations to be performed on them. The data lake can efficiently ingest and store data of every type, including not only relational data (which it persists in a columnar object format), but also semi-structured (text, logs, documents) and multi-structured (files of any type) data.

Object storage: As the foundation of the lakehouse stack, object storage comprises an even more basic abstraction layer: a performant and cost-effective means of provisioning and scaling

storage, on-demand storage.

Again, for this to work, data lakehouse architecture must be more (or less) loosely coupled, at least in comparison with the classic data warehouse, which depends on the RDBMS to provide most functions.

So, for example, several providers market cloud SQL query services that, when combined with cloud data lake and object storage services, can be used to create the data lakehouse. Think of this as the "ideal" data lakehouse — ideal in the sense that it is a rigorous implementation of a formal, loosely coupled architectural design. The SQL query service runs against the data lake service, which sits on top of an object storage service. Subscribers instantiate prebuilt queries, views and data modeling logic in the SQL query service, which functions like a semantic layer. Voila: the data lakehouse.

This implementation is distinct from the data lakehouse services that Databricks, Dremio and others market. These implementations are usually coupled to a specific data lake implementation, with the result that deploying the lakehouse means, in effect, deploying each vendor's data lake service, too.

The formal rigor of an ideal data lakehouse implementation has one obvious benefit: It is notionally easier to replace one type of service (for example, a SQL query) with an equivalent commercial or open source service. As we shall see, however, there are advantages to a less loosely coupled data lakehouse implementation, especially in connection with demanding data warehousing workloads.

What Is New And/or Different About the Data Lakehouse?

It all starts with the data lake. Again, the data lakehouse is a higher-level abstraction superimposed over the data in the lake. The lake usually comprises several zones, the names and purposes of which vary according to implementation. At a minimum, these consist of the following:

- One or more ingest or landing zones for data
- One or more staging zones, in which experts work with and engineer data
- One or more "curated" zones, in which prepared/engineered data is made available for access.

"The data lakehouse is a higher-level abstraction superimposed over the data in the lake."

In most cases, the data lake is home to all of an organization's useful data. This data is already there. So, the data lakehouse begins with an innocuous idea: Why not query against this data where it lives?

It is in the curated zone of the data lake that the data lakehouse itself lives, although it is also able to access and query against data that is stored in the lake's other zones. In this way, its proponents claim, the data lakehouse is able to support not only traditional data warehousing use cases, but also novel use cases such as data science and machine learning/artificial intelligence engineering.

The following is a mostly uncritical summary of the claimed advantages of the data lakehouse.

Subsequent parts of this report will explore and assess the validity of these claims:

More agile and less fragile than the data warehouse

Advocates argue that querying against data in the lake eliminates the multi-step process entailed in moving the data, engineering it and moving it again prior to loading it into the warehouse. (In extract, load, transform [ELT], data is engineered in

the warehouse itself. This obviates a second data movement operation.) This process is closely associated with the use of extract, transform, load (ETL) software. With the data lakehouse, instead of modeling data twice — first, during the ETL phase, and, second, to design denormalized views for a semantic layer, or to instantiate data modeling and data engineering logic in code — experts need only perform this second modeling step.

The end result is less complicated (and less costly) ETL, as well as a less fragile data lakehouse.

Query against data in situ in the data lake

Proponents say that querying against the data lakehouse makes sense because all of an organization's business-critical data is already there — that is, in the data lake. Data gets vectored into the lake from sensors and other signalers, from cloud apps and services, from online transaction processing systems, from subscription feeds, and so on.

"The data lakehouse sits atop the data lake, which ingests, stores and manages data of every type."

The strong claim is that the extra ability to query against data in the whole of the lake — that is, its staging and/or non-curated zones — can accelerate data delivery for time-sensitive use cases. A related claim is that it is useful to query against data in the lakehouse, even if an organization already has a data warehouse, at least for some time-sensitive use cases or practices.

The weak claim is that the lakehouse is a suitable replacement for the data warehouse. The third part of this report assesses the case for the lakehouse as a warehouse replacement.

Query against relational, semi-structured and multi-structured data

The data lakehouse sits atop the data lake, which ingests, stores and manages data of every type. Moreover, the lake's curated zone need not be restricted solely to relational data: Organizations can store and model time series, graph, document and other types of data there. Even though this is possible with a data warehouse, lakehouse proponents concede, it is not usually cost-effective.

More rapidly provision data for time-sensitive use cases

Expert users — say, scientists working on a clinical trial — can access raw trial results in the data lake's non-curated ingest zone, or in a special zone created for this purpose. This data is not provisioned for access by all users; only expert users who understand the clinical data are permitted to access and work with it. Again, this and similar scenarios are possible because the lake functions as a central hub for data collection, access and governance. The necessary data is already there, in the data lake's raw or staging zones, "outside" the data lakehouse's strictly governed zone. The organization is just giving a certain class of privileged experts early access to it.

Better support for DevOps and software engineering

Unlike the classic data warehouse, the lake and the lakehouse expose a variety of access APIs, in addition to a SQL query interface.

For example, instead of relying on ODBC/JDBC interfaces and/or ORM techniques to acquire and transform data from the lakehouse — or using ETL software that mandates the use of its own tool-specific programming language and IDE design facility — a software engineer can use her preferred dev tools and cloud services, so long as these are also supported by her team's DevOps toolchain. The data lake/lakehouse, with its diversity of data

exchange methods, its abundance of co-local compute services, and, not least, the access it affords to raw data, is arguably a better "player" in the DevOps universe than is the data warehouse. In theory, it supports a larger variety of use cases, practices and consumers — especially expert users.

True, most RDBMSs, especially cloud PaaS RDBMSs, now support access via RESTful APIs and/or language-specific SDKs. This does not change the fact that some experts, particularly software engineers, are not — at all — enamored of the RDBMS.

Another consideration is that the data warehouse, especially, is a strictly governed repository. The data lakehouse imposes its own governance strictures, but the lake's other zones can be less strictly governed. This makes the combination of the data lake + data lakehouse suitable for practices and use cases that require time-sensitive, raw, lightly prepared, etc., data (such as ML engineering).

Support more and different types of analytic practices

For expert users, the data lakehouse simplifies the task of accessing and working with raw or semi-/multi-structured data.

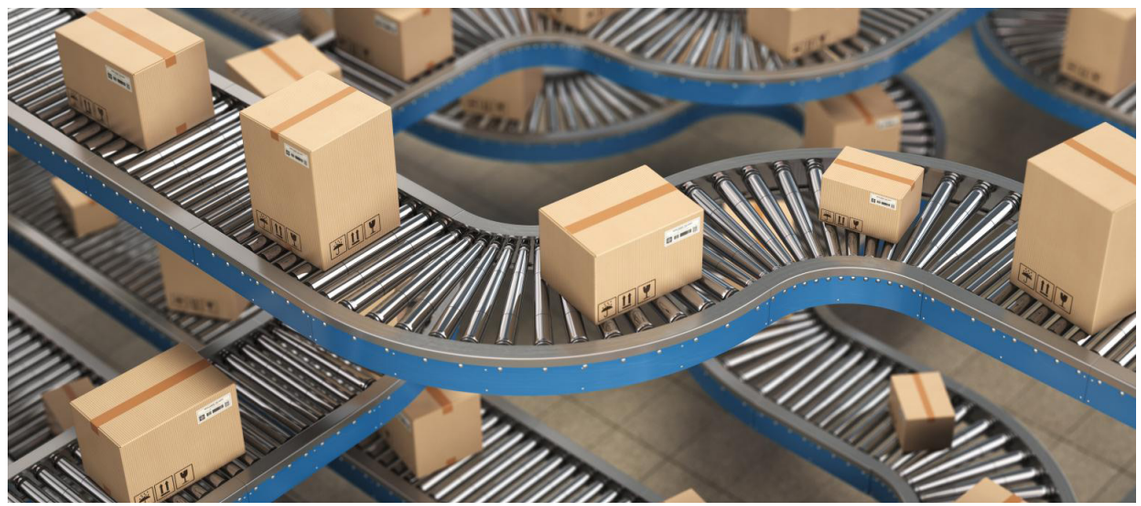
Data scientists, ML and AI engineers, and, not least, data engineers can put data into the lake, acquire data from it, and take advantage of its co-locality with an assortment of intra-cloud compute services to engineer data. Experts need not use SQL; rather, they can work with their preferred languages, libraries, services and tools (notebooks, editors and/or favorite CLI shells). They can also use their preferred conceptual vocabularies. So, for example, experts can build and work with data pipelines, as distinct to designing ETL jobs. In place of an ETL tool, they can use a tool such as Apache Airflow to schedule, orchestrate and monitor workflows.

Final Thoughts

It is impossible to disentangle the value and usefulness of the data lakehouse from that of the data lake. In theory, the combination of the two — that is, the data lakehouse layered atop the supervening data lake — outstrips the usefulness, flexibility and capabilities of the data warehouse. The discussion above sometimes refers separately to the data lake and to the data lakehouse. What is usually implied, however, is the co-locality of the data lakehouse with the data lake — the "data lake/house," if you like.

So much for the claimed advantages of the data lake. Part 2 of this report makes the case that data lakehouse architecture comprises a radical break with classic data warehouse architecture. ■

Part 2: Comparing the Data Lakehouse with the Conventional Data Warehouse



Now I focus on how the architecture of the data lakehouse compares with that of the classic, or conventional, data warehouse. This part imagines data lakehouse architecture as an attempt to implement some of the core requirements of data warehouse architecture in a modern design based on cloud-native concepts, technologies and methods. It explores the advantages of cloud-native design, beginning with the ability to dynamically provision resources in response to specific events, predetermined patterns and other triggers. It likewise explores data lakehouse architecture as its own thing — that is, as an attempt to address new or different types of practices, use cases and consumers.

How Data Lakehouse Architecture Differs from Data Warehouse Architecture

In an important sense, data lakehouse architecture is an effort to adapt the data warehouse — and its architecture — to cloud and, at the same time, to address a much larger set of novel use cases, practices and consumers. This is a less counterintuitive, and less daunting, claim than it might seem.

Think of data warehouse architecture as akin to a technical specification: It does not tell you how to design or implement the data warehouse; rather, it enumerates and describes the set of requirements (that is, features and capabilities) that the ideal data warehouse system must address. For all intents and purposes, then, designers are free to engineer their own novel implementations of the warehouse, which is what Joydeep Sen Sarma and Ashish Thusoo attempted to do with Apache Hive, a SQL interpreter for Hadoop, or what Google did with BigQuery, its NoSQL query-as-a-service offering.

The data lakehouse is an example in kind. In fact, to the extent that a data lakehouse implementation addresses the set of requirements specified by data warehouse architecture, it is a data warehouse.

In the first part of this report, we saw that data lakehouse architecture eschews the monolithic design of classic data warehouse implementations, as well as the more tightly coupled designs of big data-

era platforms, such as Hadoop+Hive, or platform-as-a-service (PaaS) warehouses, such as Snowflake.

Design-wise, then, data lakehouse architecture is quite different. But how is it different? And why?

Adapting Data Warehouse Architecture to Cloud

The classic implementation of data warehouse architecture is premised on a set of dated expectations, particularly with respect to how the functions and resources that comprise the warehouse are to be instantiated, connected together and accessed. For one thing, early implementers of data warehouse architecture expected that the warehouse would be physically instantiated as an RDBMS and that its components would connect to one another via a low-latency, high-throughput bus. Relatedly, they expected that SQL would be the sole means of accessing and manipulating data in the warehouse.

A second expectation was that the data warehouse was to be online and available at all times. Moreover, its constituent functions were expected to be tightly coupled to one another. This was a feature, not a bug, of its instantiation in an RDBMS. This made it impracticable (and, for all intents and purposes, impossible) to scale the warehouse's resources independently of one another.

Neither of these expectations is true in the cloud, of course. And we are all quite familiar with the cloud as a metaphor for virtualization — that is, the use of software to abstract and define different types of virtual resources — and for the scale-up/scale-down elasticity that is cloud's defining characteristic.

But we tend to spend less time thinking about cloud as a metaphor for the event-driven provisioning of virtualized hardware — and, by implication, for an ability to provision software in response to events, too.

This on-demand dimension is arguably the most important practical benefit of cloud's elasticity. It is also one of the most obvious differences between the data lakehouse and the classic data warehouse.

The Data Lakehouse as Cloud-native Data Warehouse

Event-driven design at this scale presupposes a fundamentally different set of hardware and software requirements. It is this event-driven dimension that cloud-native software engineering concepts, technologies and methods evolved to address. In place of tightly coupled, monolithic applications that expect to run atop always-on, always-available, physically instantiated hardware resources, cloud-native design presupposes an ability to provision discrete software functions (which developers instantiate as loosely coupled services) on-demand, in response to specific events. These loosely coupled services correspond to the constituent functions of an application. Applications themselves are composed of loosely coupled services, as with the data lakehouse and its layered architecture.

"The classic implementation of data warehouse architecture is premised on a set of dated expectations, particularly with respect to how the functions and resources that comprise the warehouse are to be instantiated, connected together and accessed."

What makes the data lakehouse cloud-native? It is cloud-native to the degree that it decomposes most, if not all, of the software functions that are implemented in data warehouse architecture. There are six functions:

- One or more functions capable of storing, retrieving and modifying data

• One or more functions capable of performing different types of operations (such as joins) on data

• One or more functions exposing interfaces that users/jobs can use to store, retrieve and modify data, as well as to specify different types of operations to perform on data

• One or more functions capable of managing/enforcing data access and integrity safeguards

• One or more functions capable of generating or managing technical and business metadata

• One or more functions capable of managing and enforcing data consistency safeguards, as when two or more users/jobs attempt to modify the same data at the same time, or when a new user/job attempts to update data that is currently being accessed by prior users/jobs.

With this as a guideline, we can say that a "pure" or "ideal" implementation of data lakehouse architecture would consist of the following components:

The lakehouse service itself. In addition to SQL query, the lakehouse might provide metadata management, data federation and data cataloging capabilities. In addition to its core role as a query service, the lakehouse doubles as a semantic layer: It creates, maintains and versions modeling logic, such as denormalized views that get applied to data in the lake.

The data lake. At minimum, the data lake provides schema enforcement capabilities, along with the ability to store, retrieve, modify and schedule operations on objects/blobs in object storage. The lake usually provides data profiling and discovery, metadata management, and data cataloging capabilities, along with data engineering and, optionally, data federation capabilities. It enforces access and data integrity safeguards across each of its constituent zones. Ideally, it also generates and manages technical metadata for the data in these zones.

An object storage service. It provides a scalable, cost-effective storage substrate. It also handles the brute-force work of storing, retrieving and modifying the data stored in file objects.

With that said, there are different ways to implement the data lakehouse. One pragmatic option is to fold all these functions into a single omnibus platform — a data lake with its own data lakehouse. This is what Databricks, Dremio and others have done with their data lakehouse implementations.

Why Does Cloud-native Design Matter?

This invites some obvious questions. First, why do this? What are the advantages of a loosely coupled architecture vs. the tightly integrated architecture of the classic data warehouse? As we have seen, one benefit of loose coupling is an ability to scale resources independently of one another — to allocate more compute without also adding storage or network resources. Another benefit is that loose coupling eliminates some of the dependencies that can cause software to break. So, for example, a change in one service will not necessarily impact, let alone break, other services. Similarly, the failure of a service will not necessarily cause other services to fail or to lose data. Cloud-native design also uses mechanisms (such as service-orchestration) to manage and redress service failures.

Another benefit of loose coupling is that it has the potential to eliminate the types of dependencies that stem from an implementation's reliance on a specific vendor's or provider's software. If services communicate and exchange data with one another solely by means of publicly documented APIs, it should be possible to replace a service that provides a definite set of functions (such as SQL query) with an equivalent service. This is the premise of pure or ideal data lakehouse architecture: Because each of its constituent components is, in effect, commoditized (such that equivalent services are available from all of the major cloud infrastructure providers, from third-party SaaS and/or PaaS providers, and as open source offerings), the risk of provider-specific lock-in is reduced.

The Data Lakehouse as Event-driven Data Warehouse

Cloud-native software design also expects that the provisioning and deprovisioning of the hardware and software resources that power loosely coupled cloud-native services is something that should happen automatically. In other words, to provision a cloud-native service is to provision its enabling resources; to terminate a cloud-native service is to deprovision these resources. In a sense, cloud-native design wants to make hardware — and, to a degree, software — disappear, at least as a variable in the calculus of deploying, managing, maintaining and, especially, scaling business services.

From the viewpoints of consumers and expert users, there are only services — that is, tools that do things.

For example, if an ML engineer designs a pipeline to extract and transform data from 100 GBs of log files, a cloud-native compute engine dynamically provisions ***n*** compute instances to process her workload. Once the engineer's workload finishes, the engine automatically terminates these instances.*

Ideally, neither the engineer nor the usual IT support people (DBAs, systems and network administrators, and so on) need to do anything to provision these compute instances or, crucially, the software and hardware resources on which they depend. Instead, this all happens automatically — in response, for example, to an API call initiated by the engineer. The classic, on-premises data warehouse was just not conceived with this kind of cloud-native, event-driven computing paradigm in mind.

The Data Lakehouse as Its Own Thing

The data lakehouse is, or is supposed to be, its own thing. As we have seen, it provides the six functions listed above. But it depends on other services — namely, an object storage service and, optionally, a data lake service — to provide basic data storage and core data management functions. In addition, data lakehouse architecture implements a novel set of software functions that have no obvious parallel in classic data warehouse architecture. In theory, these functions are unique to the data lakehouse.

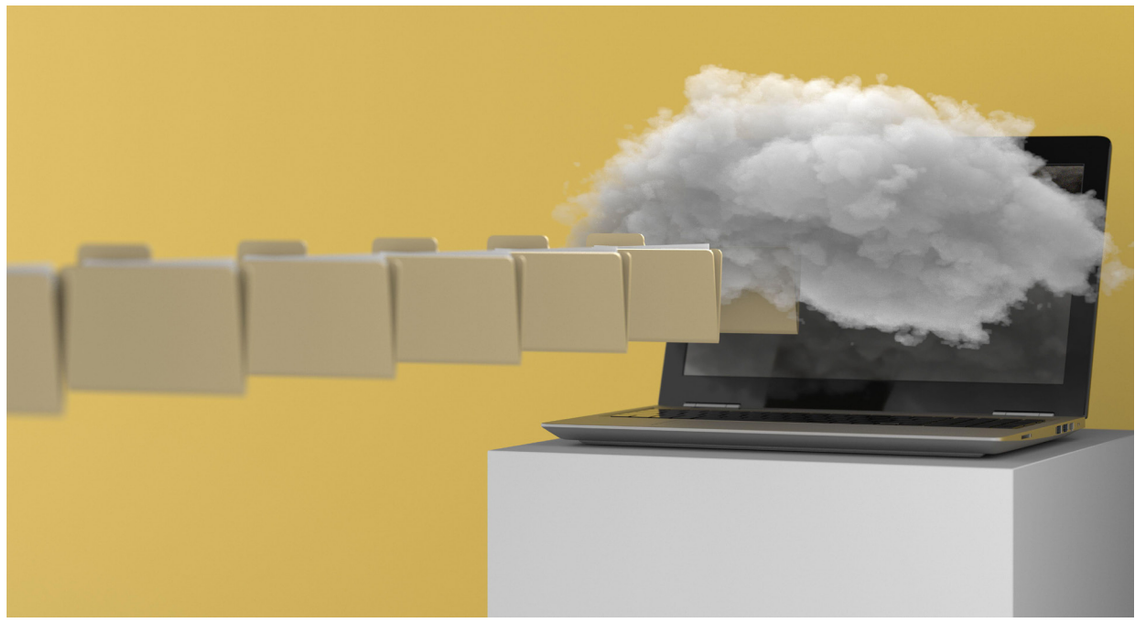
These are:

- One or more functions capable of accessing, storing, retrieving, modifying and performing operations (such as joins) on data stored in object storage and/or third-party services. The lakehouse simplifies access to data in Amazon S3, AWS Lake Formation, Amazon Redshift, and so on.
- One or more functions capable of discovering, profiling, cataloging and/or facilitating access to distributed data stored in object storage and/or third-party services. For example, a modeler creates n denormalized views that combine data stored in both the data lakehouse and in the staging zone of an AWS Lake Formation (that is, a data lake). The modeler also designs a series of more advanced models that incorporate data from an Amazon Redshift sales data mart.

In this respect, however, the lakehouse is not actually all that different from a PaaS data warehouse service. The fifth and final part of this report will explore this similarity in depth. ■

** The software required to make this work is still very new. Arguably, some of it does not yet exist — at least in a sense analogous to the RDBMS, whereby, for example, a query optimizer parses each SQL query, estimates the cost of running it and pre-allocates the necessary resources. This is not magic; rather, it is grounded in the rigor of mathematics. Under the covers, the RDBMS' query optimizer uses relational algebra to translate SQL commands into relational operations. It creates a query plan — that is, an optimized sequence of these operations — that the database engine uses to allocate resources to process the query. This proactive model is quite different from the reactive model that predominates in cloud-native software. For example, cloud-native design principles might expect to use real-time feedback from observable components to determine the cost of running a workload and provision sufficient resources. To the degree that cloud-native software has a proactive dimension, this comes via pre-built rules or ML models. Workloads are not proactively soluble via math.*

Part 3: Assessing the Viability of the Data Lakehouse



Now, let's explore the advantages and disadvantages of the data lakehouse — both as its own thing and as a replacement for the data warehouse. Part 3 considers the requirements the data lakehouse must address if it is to completely replace the warehouse.

The Formal, Technical Requirements of Data Warehouse Architecture

First, what must the data lakehouse be able to do if it is fully to replace the data warehouse? At minimum, the set of capabilities provided by its software functions must also comply with the formal, technical requirements of data warehouse architecture.

From the point of view of data warehouse architecture, it is less important that a query platform should provide fast results than that these results should be uniform and replicable.* In practice, the trick is to balance these requirements against one another, to achieve query results that are fast enough and also uniform and replicable.

This is actually much easier said than done. It is the reason why Hive + Hadoop consistently failed when used as a data warehouse replacement. It is the reason why distributed NoSQL systems almost always have problems when tapped for use as would-be RDBMS or data warehouse replacements.

Against this backdrop, let's review the formal, technical requirements of data warehouse architecture.

The data warehouse is the following:

- A single, central repository for topical and historical business data
- A system that permits a panoptic view across the business and its function areas
- A system that permits a monitoring/feedback loop into the performance of the business
- A system that consumers can use to pose common and/or unpredictable (ad hoc) questions
- A system that delivers consistent, uniform query results (i.e., everybody has the same data)
- A system capable of hosting concurrent jobs/users and highly demanding mixed workloads
- A system capable of enforcing strict, invariant data management and data processing controls
- A system capable of anticipating and resolving conflicts that occur between

requirements 5, 6 and 7

Does the data lakehouse fit the bill? It depends on how you implement data lakehouse architecture.

If you architect your lakehouse by pointing a SQL query service at the curated zone of a data lake, you will have an implementation that is almost certainly able to scale to address requirements 1 through 4. However, this implementation will probably struggle with requirements 5 through 8, inasmuch as each demands an engine that has the ability to enforce strict consistency, uniformity and replicability guarantees while also performing multiple, simultaneous operations on data. This is because requirements 5 through 8 go to the wicked problem of managing (and resolving) the conflicts that occur as a result of concurrency.

Reality Check: It Is Not OK To Drop ACID-like Safeguards

In a classic, tightly coupled data warehouse implementation, the warehouse is usually instantiated in a relational database, or RDBMS. Almost all RDMBSs enforce ACID safeguards that enable them to perform concurrent operations on data while at the same time maintaining strong consistency.

"In a classic, tightly coupled data warehouse implementation, the warehouse is usually instantiated in a relational database, or RDBMS."

In the popular imagination, ACID safeguards are associated with online transaction processing (OLTP), which is also strongly associated with the RDBMS. It must be emphasized, however, that a data warehouse is not an OLTP system: You do not need to deploy a data warehouse on an RDBMS.

Reduced to a primitive technology prescription, the database engine at the heart of the data warehouse requires two things: a data store that has the ability to create and manage tables, as well as to append records to them, and logic to resolve the conflicts that tend to occur between concurrent operations on data. So, it is possible to design the data warehouse as an append-only data store and to commit new records in a timeline — for example, as new rows. If you can only append new records (that is, you cannot change or delete existing ones), concurrency conflicts cannot occur. You can likewise design coordination logic to address data uniformity requirements, to ensure that if n users/jobs query the warehouse at the same time, each will query against exactly the same records in the timeline.

In practice, however, the easiest way to address these requirements is by using an RDBMS. The RDBMS is also optimized to quickly and correctly perform the relational operations (such as different kinds of joins) that are essential in analytical work. These are but two of the reasons the on-premises data warehouse is usually identical with the RDBMS — and why a procession of would-be replacements, such as Hadoop + Hive, failed to displace the conventional warehouse.

It is also the reason almost all PaaS data warehouse services are designed as RDBMS-like systems. As I wrote in a previous article of mine, "if you eschew in-database ACID database safeguards, you must either roll your own ACID enforcement mechanisms or accept data loss as inevitable." This means that you have a choice among building ACIDic logic into your application code, designing and maintaining your own ACID-compliant database, or delegating this task to a third-party database.

Data Warehouse Workloads Require Consistency, Uniformity and Replicability Safeguards

Like it or not, consistency, uniformity and replicability are common requirements for production data warehouse workloads.

For example, core operational business workflows routinely query the warehouse. In a production use case, then, the data lakehouse that replaces it might be expected to service hundreds of such queries per second.

Let's consider what this entails by looking closely at a representative workload — for example, a credit-application process that queries the lakehouse for credit-scoring results dozens of times each second. For statutory and regulatory reasons, queries processed at the same time must return correct results ("correct" as in they all use the same scoring model). This model also uses the same point-in-time data, albeit adjusted for customer-specific variations.

But what if a concurrent operation attempts to update the data used to feed one or more of the model's parameters? The RDBMS' ACID safeguards would prevent this update from being committed until after the results of the (dependent) credit-scoring operations had first been committed.

Can a SQL query service enforce the same safeguards? Can it do this even if objects in the data lake's curated zone are accessible to other services (say, an AWS Glue ETL service), which are also able to update data at the same time?

The example above is by no means uncommon. It is, rather, the routine way of doing things. In other words, if you need to ensure consistent, uniform and replicable results, you need ACIDic safeguards. The upshot is that data warehouse workloads require these safeguards.

Can Data Lakehouse Architecture Enforce These Safeguards?

It depends. The first issue has to do with the fact that it is difficult to coordinate dependent operations across loosely coupled services. So, for example, how does an independent SQL query service restrict access to records in an independent data lake service? This is necessary to prevent concurrent users from modifying objects in the lake's curated zone.

"Like it or not, consistency, uniformity and replicability are common requirements for production data warehouse workloads."

In the tightly coupled RDBMS, the database kernel manages this. In the credit-scoring example above, the RDBMS locks the rows in the table(s) in which dependent data is recorded; this prevents other operations from modifying them. It is just not clear how to manage this in the data lakehouse architecture, with its layered stack of decoupled services.

A fit-for-purpose data lakehouse service should be able to enforce ACID-like safeguards — if it is its own data lake and can control concurrent access to (and modification of) the objects in its data lake layer. Again, this is what Databricks and Dremio have done in their implementations of data lakehouse architecture. They solve the problem of coordinating concurrent access to (and operations on) resources that are shared across services by less loosely coupling these services to one another.

By contrast, this is much more difficult if the data lakehouse is implemented as a layered stack of loosely coupled, independent services — for example, a discrete SQL query service that sits atop the curated zone of a discrete data lake service, which itself sits atop a discrete object storage service. It cannot achieve strong consistency because it cannot control access to the objects in the data lake.

Final Thoughts

In any distributed topology, the challenge is to coordinate concurrent access to shared resources while at the same time managing

multiple dependent operations on these resources across space and time. This is true whether software functions (and the resources they operate on) are tightly or loosely coupled. So, for example, the way distributed processing is handled in a classic data warehouse implementation is by instantiating the warehouse as a massively parallel processing (MPP) database. The MPP database kernel identifies, schedules and coordinates dependent operations across the nodes of the MPP cluster, as well as manages (and resolves) the conflicts that occur between dependent operations. In other words, the MPP database kernel is able to enforce strict ACID safeguards while also performing concurrent operations in a distributed topology. This is no mean feat.

By contrast, a loosely coupled distributed software architecture, such as data lakehouse architecture, is confronted with the problem of coordinating access to resources and managing dependencies across what are, in effect, independent services. This is a wicked problem.

This is one reason the data lakehouse (like the data lake itself) usually functions as what is called an eventually consistent platform rather than as a strongly consistent platform.

On the one hand, it is able to enforce ACID-like safeguards; on the other hand, it may lose data and be unable to deliver uniformly replicable results. To enforce strict ACID safeguards would require combining the data lakehouse and the data lake into one platform: to tightly couple both services to one another.** For what it is worth, this is the likely tendency of data lake/lakehouse evolution: Each will converge into the other, provided the idea of the data lakehouse actually has staying power.

However, to implement the data lakehouse as its own data lake is (in effect) to recapitulate the phylogeny of the data warehouse: It is to put the data lakehouse on the same evolutionary path as the warehouse

itself. At a minimum, it is to tightly couple the lakehouse and the lake, to introduce (and to consolidate) a dependency on a single software platform and, just as important, on a single provider. ■

** For more on the distinction between uniformity and replicability, read this article. The nickel summary: If n consumers query the same data at the same time, they should all receive exactly the same results, even if another consumer (e.g., an ETL job) attempts to update this data. This is what is meant by uniformity. If the data warehouse were to re-sequence and re-run these operations in times, it would produce exactly the same results each time. This is what is meant by replicability.*

*** I will not attempt to vet the claims to strong consistency made by prominent data lakehouse + data lake providers. It seems to me that theirs is an especially wicked problem indeed. Instead, I say with Augustine: "I believe that I may understand."*

Part 4: The Use of Data Modeling with the Data Lakehouse

Now we'll look at the role of data modeling in the context of designing, maintaining and using the lakehouse. I assess the claim that the lakehouse is a light alternative to the data warehouse.

Data Lakehouse vs. Data Warehouse: Death, Taxes and Data Modeling

In making the case for the lakehouse as a data warehouse replacement, proponents usually point to a few extra benefits. The first benefit is that the lakehouse is supposed to simplify data modeling, which (in turn) is supposed to simplify ETL/data engineering. A second claimed benefit goes to the reduced cost of managing and maintaining ETL code. A third claimed benefit is that the elimination of data modeling makes the lakehouse less apt to "break" — that is, the routine alterations and

vicissitudes of business, such as merger-and-acquisition activity, expansion into (or retreat from) new regions, and the launching of new services, do not break the lakehouse's data model because there is no data model to break.

How a Bill Becomes a Law, or How Data Is, or Isn't, Modeled for the Data Lakehouse

To understand what this means, let's look at a best-case scenario for modeling in the data lakehouse.

- Data is ingested by the data lake's landing zone.
- Some/all raw data is optionally persisted in a separate zone for archival storage.
- Raw data, or predefined extracts of raw data, are moved into one of the data lake's staging zones. The lake may maintain separate staging zones for different types of users/practices.
- Raw OLTP data may undergo immediate data engineering (for example, scheduled batch ETL transformations) after which it gets loaded directly into the data lake's curated zone.
- Data in the lake's staging zones is made available to different kinds of jobs/expert users.
- A subset of data in the lake's staging zones is engineered and moved into the curated zone.
- Data in the curated zone is lightly modeled — for example, it is stored in an optimized columnar format.
- The data lakehouse is a modeling overlay (akin to a semantic model) that is superimposed over data in the lake's curated zone or, optionally, over select data in its staging zones.
- Data in the lake's curated zone is unmodeled. In the data lakehouse, per se, data modeling is instantiated in application- or use case-specific logical models, akin to denormalized views.

So, for example, rather than engineering data so it can be stored in and managed by a data warehouse (usually an RDBMS), data is lightly engineered — put into a columnar format — prior to its instantiation in the data lake's curated zone. This is where the data lakehouse is supposed to take over.

How much data must be instantiated in the lakehouse's curated zone?

The simple answer is as little or as much as you want. The pragmatic answer is it depends on the use cases, practices and consumers the data lakehouse is intended to support. The nuanced answer: Modeling data at the level of a historical data store (such as the warehouse or the lakehouse) has an essential strategic purpose, too.

Before we unpack this claim, let's look at what happens to data once it gets loaded into the data lake's curated zone. The data in the curated zone is usually persisted in a columnar format, such as Apache Parquet. This means, for example, that the volume of data that comprises the curated zone is distributed across hundreds, thousands, even millions of Parquet objects, all of which live in object storage. This is one reason the curated zone usually eschews a complex data model in favor of a flat or one-big-table (OBT) schema — basically, a scheme in which all data is stored in a single denormalized table. (The idea is to maximize the advantages of object storage — high-bandwidth and sustained throughput — while minimizing the cost of its high and/or unpredictable latency.) A claimed benefit of this is that the flat-table or OBT schema eliminates the need for the logical data modeling that is usually performed in 3NF or Data Vault modeling, as well as the dimensional data modeling performed in Kimball-type data warehouse design. This is a significant timesaver, lakehouse proponents say.

But wait, isn't this how data is also modeled in some data warehouse systems?

One problem with this is that data warehouse systems commonly run flat-table and OBT-type schemas, too. In fact,

OBT schemas were used with the first data warehouse appliance systems in the early 2000s. Today, OBT schemas are commonly used with cloud PaaS data warehouses, such as Amazon Redshift and Snowflake. The upshot is that if you do not want to perform heavy-duty data modeling for the data warehouse, you do not have to. For good or ill, plenty of organizations opt not to model.

This gets at a more perplexing problem, however: Why do we model data for the warehouse in the first place? Why do data management practitioners place such great store in data modeling?

The reason is, like it or not, data modeling and engineering are bound up with the core priorities of data management, data governance and data reuse. We model data to better manage, govern and (a function of both) reuse it. In modeling and engineering data for the warehouse, we want to keep track of where the data came from and what has been done to it, when, and (not least) by whom or by what. (In fact, the ETL processes used to populate the data warehouse generate detailed technical metadata to this effect.) Similarly, we manage and govern data so that we can make it available to, and discoverable by, more and different types of consumers — and, especially, by non-expert consumers.

To sum up, we model data so we can understand it, so we can impose order on it, and so we can productionize it in the form of managed, governed, reusable collections of data. This is why data management practitioners tend to be adamant about modeling data for the warehouse. As they see it, this emphasis on engineering and modeling data makes the warehouse suitable for a very wide range of potential applications, use cases and consumers.

This is in contrast to alternatives that focus on engineering and modeling data for a semantic layer, or encapsulating data engineering and modeling logic in code itself. These alternatives tend to focus on specific applications, use cases and consumers.

The Unbearable Fragility of Data Modeling

A final problem is that the typical anti-data modeling frame is misleading. To eschew modeling at the data warehouse/lakehouse layer is to concentrate data modeling in another layer. You are still modeling and engineering data; you are just doing it in different contexts, such as in a semantic layer or in code itself. You still have code to maintain. You still have things that can (and will) break.

Imagine, for example, that a business treats Europe, the Middle East and Africa (EMEA) as a single region, then suddenly decides to create separate EU, ME and Africa divisions. Yes, a change of this kind will require it to make changes to its data warehouse's data model. However, it will also affect the denormalized views instantiated in its semantic layer. At minimum, modelers and business subject-matter experts must refactor these views. They may also opt to rebuild some of them from scratch.

The essence of the claim is that it is easier, faster and cheaper to fix problems in a semantic layer or in code than to make changes to a central repository, be it a data warehouse or a data lakehouse. This claim is not wrong, exactly, just tendentious. At the very least, it arises out of a distorted sense of how and why data gets modeled, be it for the old-school data warehouse or for the data lakehouse.

Both sides have valid concerns and make good points. It is a question of balancing costs and benefits.

Final Thoughts

To assume that the lakehouse reduces or eliminates data modeling — and, with it, the complexity of ETL engineering — is to

ignore the purpose of data modeling in data management. It is likewise to play a kind of shell game with ETL. As a friend who works as an ML solutions architect likes to remind me: "You can never escape the work of ETL; you can only ever push it somewhere else."

It is never easy to accommodate business change. To change something about the business is to break the correspondence between a data model that represents events and phenomena in the business's world and reality itself. This correspondence is never identical.

At best, it serves a purpose: It makes it possible for the business to see, understand and manipulate its structure, operations, and so on.

It is probably easier, on the whole, to shift most data modeling logic to a BI/semantic layer. In the scenario above, for example, modelers and SMEs must design a new warehouse data model; repopulate the data warehouse; and fix and identify broken queries, stored procedures, UDFs, and so on. However, they must also fix the modeling logic that is instantiated in the BI/semantic layer. This is extra work, no doubt about it.

"To change something about the business is to break the correspondence between a data model that represents events and phenomena in the business's world and reality itself."

As I have shown, however, this is not a problem that is specific to the data warehouse. In fact, it has just as much salience for any organization that implements a data lakehouse system. The idea of a lightly modeled historical repository for business data is not new; ergo, if you want to eschew modeling for the data lakehouse or the data warehouse, that is an option — and has been for quite some time.

On the other hand, an organization that opts to model data for its lakehouse should have less modeling to do in its BI/semantic layer. Perhaps much less. Yet, the data in this lakehouse should be lucid and understandable to, as well as reusable by, a larger pool of potential consumers.

For this reason, these consumers are also more likely to trust the data in the data lakehouse.

By the way, this is another case in which a less loosely coupled data lakehouse implementation (for example, Databricks' Delta Lake or Dremio's SQL Lakehouse Platform) has an advantage relative to what I call an "ideal" data lakehouse implementation — that is, an implementation in which the data lakehouse is cobbled together out of loosely coupled services, such as a SQL query service, a data lake service or a cloud object storage service. It makes more sense to model and govern data in a tightly coupled data lakehouse implementation in which the data lakehouse has sovereign control over business data.

It is not clear how this is practicable in an implementation in which a SQL query service does not have sovereign control over the objects that live in the curated zone of the underlying data lake. ■

Part 5: The Data Lakehouse vs. the PaaS Data Warehouse

For the most part, previous parts of this report have treated the data warehouse as a kind of strawman. For example, I have mostly compared the data lakehouse and its cloud-native design with classic implementations of the data warehouse — as if cloud-native concepts and methods had not also been applied to data warehouse architecture or, put another way, as if data warehouse architecture were stuck in time.

So far, I have said next to nothing about the platform-as-a-service (PaaS) or query-as-a-service (QaaS)* data warehouse, nor have I discussed these schemes as novel implementations that are comparable in both capabilities and (as it were) cloud "nativity" with the no less novel data lakehouse.

This idea is implicit in prior analysis, however. In Part 2, for example, I made the point that data warehouse architecture is less a technology prescription than a technical specification: Instead of telling us how to build the data warehouse, it tells us what that system is supposed to do and how it is supposed to behave. It outlines the features, capabilities, and so on that are required by that system.

One implication of this is that there are several possible ways to implement the data warehouse. Another is that the requirements of data warehouse architecture are not necessarily in conflict with those of cloud-native design. A final implication is that the cloud-native data warehouse actually has quite a few things in common with the data lakehouse, even as it departs from that implementation in critical respects.

With this as background, let's pivot to the culminating questions of this report: What do the data lakehouse and the PaaS data warehouse have in common, and how are they different?

The PaaS Data Warehouse Sure Looks Like a Data Lakehouse

Both the PaaS data warehouse and the data lakehouse have quite a bit in common. Like the data lakehouse, the PaaS data warehouse:

- Lives in the cloud
- Separates compute, storage and other resources
- Can elastically expand/contract to suit

spikes in demand, seasonal/one-off use cases, and so on

- Is event-driven and can provision compute or (if necessary) temporary storage resources in response to event triggers and deprovision these resources once they are no longer needed
- Is co-local with (intra-cloud adjacent to) other cloud services, including the data lake
- Like the data lake/house, writes data to and reads data from inexpensive cloud object storage
- Can query/facilitate federated access to data stored in any of the data lake's zones
- Can eschew all but basic data modeling, as in flat or OBT schemas
- Can ingest, manage and perform operations on semi- and multi-structured data
- Can query across multiple data models (for example, time-series, document, graph and text)
- Can expose denormalized views (models) to support specific uses cases, applications and so on
- Exposes different types of RESTful endpoints, in addition to SQL
- Supports — via discrete APIs or language-specific SDKs — GraphQL, Python, R, Java and more.

"The 'ideal' data lakehouse implementation is cobbled together out of discrete, fit-for-purpose services."

The PaaS Data Warehouse Is (More) Tightly Coupled Where It Matters

Compared to the data lakehouse, the cloud-native data warehouse seems like a tightly coupled stack.**

The advantage of this is that the cloud-native warehouse is able to manage and control the software functions that read and write data, as well as schedule, distribute and perform operations on data; manage dependencies between these operations; implement consistency, uniformity and replicability safeguards; and so on. The cloud-native data warehouse is able to enforce strict ACID safeguards.

The "ideal" data lakehouse implementation is cobbled together out of discrete, fit-for-purpose services. So, for example, an ideal data lakehouse implementation consists of a SQL query service superimposed on top of a data lake service, which itself sits on top of a cloud object storage service. This is consistent with a trend in software design that aims to decompose large, complex programs into smaller, function-specific programs, which are instantiated as discrete services. These services are "decoupled" from one another in the sense that they have very little knowledge about how the complementary services with which they interact are supposed to work. By assembling multiple services together, you can approximate the behavior and performance of a large (monolithic) application. And, you also realize a few of the benefits that derive from this design. (For more on this, see Part 2 of this report.)

The disadvantage of this is that it poses problems vis-à-vis concurrent computing, especially when it comes to coordinating concurrent access to shared resources. As I showed in Part 3 of this report, the data warehouse solves this problem by having the RDBMS kernel enforce strict ACID safeguards.

It is not obvious how to solve this problem in an ideal data lakehouse implementation. One solution is to follow Databricks' lead — that is, to couple the data lake to the data lakehouse in a single platform.

If a data lakehouse was its own data lake — and if it could also supervise concurrent access to the data in this lake — a data lakehouse service might be able to enforce

ACID-like safeguards. However, in this model, the data lakehouse and the data lake would be tightly coupled, creating dependency on a single software platform — and a single provider.

A Data Warehouse Is a Data Warehouse … Is a Data Lakehouse?

Let's pivot to a provocative question: Can the PaaS data warehouse do all of the things the data lakehouse can do? Possibly. Think about it: What is the difference between a SQL query service that queries against data in the curated zone of a data lake and a PaaS data warehouse that lives in the same cloud context, has access to the same underlying cloud object storage service and is able to do the same thing? What is the difference between a SQL query service that facilitates access to data in the lake's archival, staging and other zones and a PaaS data warehouse that is able to do the same thing?

The data lake and the data warehouse seem to have been converging toward one another for a long time. So, on the one hand, the lakehouse looks like a textbook example of lake-to-warehouse convergence. On the other hand, the warehouse's support for multiple data models and its retrofitting with data federation and multi-structured query capabilities — that is, the ability to query files, objects or arbitrary data structures — are arguably examples of a warehouse-to-lake convergence trend.

Let's look at a few of the claimed differences between the data lakehouse and the data warehouse and see if these, too, have been obviated by convergence. Here are a few obvious ones to consider:

Has the ability to enforce safeguards to ensure the uniformity and replicability of results

I discussed this above. The PaaS data warehouse wins this one easily.

Has the ability to perform core data warehousing workloads

As I also said in Part 3 of this report, it is less important that a query platform provide fast results than that these results are uniform and replicable. In fact, the trick is to balance all three requirements against one another. This means the platform is able to achieve query results that are fast enough while at the same time maintaining a consistent state and ensuring the uniformity and applicability of results. The PaaS data warehouse is just a much faster query-processing platform than a SQL query service. It wins this one easily.

Eliminates the requirement to model and engineer data structures prior to storage

Part 4 of this report debunked this claim. The practice of modeling and engineering data structures prior to instantiating them in storage is not specific to the data warehouse; rather, it is performed to promote data intelligibility, data governance and data reuse.

The upshot is that organizations will opt to model and engineer data for both the PaaS data warehouse and the data lakehouse. Organizations that eschew all but basic data modeling for their PaaS data warehouses could realize significant improvements in query performance, relative not only to normalized (3NF) schemas but also to denormalized schemas.

Protects against cloud-service-provider lock-in

Advocates argue that ideal data lakehouse architecture helps insulate subscribers against the risk of service provider-specific lock-in. So, to cite one example, if a subscriber is dissatisfied with its existing SQL query service, it can swap this service out for another, comparable one.

How practicable is this, however? Will the modeling instantiated in the subscriber's semantic layer also transfer? Will experts have to refactor modeling logic to work with the new service? And, if a subscriber wants to switch to a different data lake — say, from

AWS Lake Formation to a fit-for-purpose data lake service, also hosted in AWS — how simple is it, really, to swap out one service for another? Quite aside from the question of data movement, there is also that of feature and function migration. For example, will developers have to refactor the data engineering and data modeling logic instantiated in their code to work with the new service?

On balance, an ideal data lakehouse implementation still has an advantage here. However, this is less true of an implementation that couples the data lakehouse to a specific lake service.

Has the ability to support a diversity of practices, use cases and consumers

The data warehouse has improved significantly in this area. However, the data lake has the advantage. It is cheaper and more convenient to ingest, store, engineer and experiment with data in the lake than it is in the data warehouse. There are fewer impediments, such as internal stumbling blocks in the form of policies, controls and processes, to constrain the use and the usefulness of the data lake in connection with data engineering, ML/AI engineering, data science and other experimental use cases. The data lake wins this one going away.

Has the ability to query against/across multiple data models

The data lakehouse sits atop the data lake, which is designed to ingest, store and manage data of any type. Ergo, if an organization puts time series, graph, document and other data into the data lake, this data will be available to, and queryable by, its data lakehouse. Right?

Usually. For example, a SQL query services such as Presto can use a connector to access, say, MongoDB, a NoSQL document database. In this scheme, Presto accesses MongoDB as if it were one or more external tables. Presto has connectors for other repositories and data producers, as well. Similarly, commercial SQL query-as-a-service providers will usually provide data source connectors. (Several of these are based on Presto, which simplifies things.)

“The data lakehouse sits atop the data lake, which is designed to ingest, store and manage data of any type.”

The real trick is to link information across data models — that is, across relational, semi-, and multi-structured data (for example, to relate CUST “Alan Smithee,” who has loyalty CARD number “8086486DX2,” in a relational data model to client “Wang Ermazi,” who lives at “2500 West End Ave” in ZIP code “37203,” in a document database). Again, this gets at the difficulty of supporting use cases that require or benefit from resource sharing in any ideal implementation in which software functions are supposed to be decomposed into discrete, decoupled services.

How does one do this in ideal data lakehouse architecture — that is, with just a SQL query service?

As for the PaaS data warehouse, it, too, can use Presto (or its own federated query services) to get at data stored in NoSQL repositories and other sources. More importantly, many relational databases can also ingest, manage, query against and perform operations on time series, graph, document, text and other data. They can link information across data models, too.

Let’s call this one a draw, although evidence suggests a multi-model RDBMS will do a better job with this if it also stores and manages the requisite time series,

graph and other data internally.

Final Thoughts: The Complementary Data Lakehouse

This is not to dismiss the data lakehouse as a useful innovation. The use cases described in Part 1 of this report are indisputably compelling. Moreover, it is arguably easier — in the sense that it is possible both to move quickly and at the same time to bypass internal impediments — to use the data lakehouse to support time-sensitive, unpredictable, one-off and other workloads and use cases.

The data warehouse is (as it should be) a strictly governed system: It does not "turn," it is not changed, on a dime. But this is to the data lakehouse's advantage, in that it comprises a less strictly governed, more agile alternative to the warehouse. In other words, the lakehouse can be seen as a complement to, and not a replacement of, the data warehouse.

The problems I explore in Part 5 and its companion parts stem from the drive to replace the data warehouse with the data lakehouse. In this specific respect, the data lakehouse falls short. The upshot is that it is difficult, if not impossible, to square the circle — to reconcile the design requirements of an ideal data lakehouse implementation with the technical requirements of data warehouse architecture. ■

** Basically, Google BigQuery, as distinct from the type of SQL query services used with the data lakehouse.*

*** Under their covers, most PaaS data warehouse services are probably architected on similar schemes. Their constituent software functions — that is, services — are typically tightly coupled, however, such that they cannot be exchanged for equivalent services. So, for example, a subscriber cannot expect to take Amazon Redshift's query optimizer and use it with Snowflake's PaaS data warehouse. Neither AWS nor Snowflake expose API endpoints to permit anything like this use case.*