O'REILLY®

The Evolving Role of the Data Engineer

Change and Continuity in Data Practices

Andy Oram

REPORT

The Evolving Role of the Data Engineer

Change and Continuity in Data Practices

Andy Oram

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

The Evolving Role of the Data Engineer
by Andy Oram

Printed in the United States of America.

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

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jon Hassell
Development Editor: Sarah Grey
Production Editor: Daniel Elfanbaum
Copyeditor: Octal Publishing, LLC
Proofreader: Abby Wheeler
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

April 2020: First Edition

Revision History for the First Edition
2020-04-14: First Release

This work is part of a collaboration between O'Reilly and Qubole. See our statement of editorial independence.

978-1-492-05248-7

[LSI]

Table of Contents

Foreword v

The Evolving Role of the Data Engineer 1
Data Engineering Today 4
Structuring Data 17
Choosing the Right Data Processing Engine 36
Development Best Practices 42
Orchestration 46
Conclusion 50

Appendix. Best Practices for Managing Resources 53

Foreword

Today's exponential growth in data velocity, volume, and variety is forcing enterprises to look for a platform that addresses their immediate needs and scales to meet their requirements. Modern organizations have data of multiple types and variable velocity, consumed by multiple personas on a continuous basis. The organizations are also accountable, legally and ethically, to their customers for how they collect and store personal information.

What's more, modern interactive analytics and machine learning demand continuous data processing as users experiment and iterate with different data types to arrive at the insights these tools can provide their businesses.

Successful companies also implement financial governance controls that lead to cost savings when taking advantage of public clouds. This reduces guesswork and avoids difficult cost-controlling exercises.

Enterprises want to do more with their data lakes: they see the benefits of leveraging inexpensive storage, the flexibility to support a great diversity of data, and advanced analytics that offer meaningful insights.

At Qubole, we've created an *Open Data Lake Platform* that addresses all of this complexity as well as potential future requirements. It supports open industry-standard ACID transactionality as a built-in feature to deliver on data privacy and right-to-be-forgotten requirements. We've also made sure to support continuous data engineering and streaming analytics in addition to data exploration, ad hoc analytics, and machine learning.

In the process of designing this solution, we've learned a lot—and you'll find those insights in the pages that follow. If you're pursuing a career in data engineering or looking for ways to adapt your enterprise to the world of big data, this report is our way of sharing the knowledge you need to find your way forward. We hope you find the world of big data engineering as intriguing as we do.

— The Qubole Marketing Team
March 2020

The Evolving Role of the Data Engineer

Every organization can benefit from data, used adeptly in coordination with the organization's goals. Today's process of accumulating enormous amounts of information from different sources—*big data* in common parlance—is like a powerful telescope that lets you see further into the universe and learn about trends you couldn't track before.

The term *data engineer* is relatively new, and the role appears only sporadically in technical literature. It is the data scientists who are exalted as today's heroes: the high-flying fighter pilots of big data who create machine learning models for predictions and other analytics. But no airplane could fly without a small army of mechanics and other trained staff to prepare the planes. In this analogy, the data engineer is like the airplane mechanic, preparing the data that enables the data scientist to carry out organizational goals. When you add up all the tasks that fit under this role—getting the data, cleaning it, creating enhanced versions—observers often claim that data engineering comprises 80 to 90% of the work organizations do with data.

Why did data engineering become so important? During the 1990s, the world underwent a momentous transformation that reached more and more deeply into our lives and brought cascading impacts upon just about every career. The transformation was driven by a hodgepodge of trends:

- New roles for data throughout society, such as the growth of the web and social media
- New sources of commercial data
- The vast speed-up in telecommunications and the internet
- New data sources such as sensors, cameras, and internet-enabled household devices
- Inexpensive storage
- Amazing new tools and algorithms for analytics, particularly machine learning

As these changes took hold, the profession of data engineering emerged to gather, store, and distribute the data. The requirements and procedures in these professions are very different from the database administrator (DBA) job that attracted so many people starting in the 1980s. These major changes have taken place during the lifetimes of even fairly young employees.

Increases in the amount and speed of data being generated throughout the world were already straining the computer industry by the 1990s, when common number prefixes such as *giga-* and *peta-* no longer sufficed, and new prefixes such as *zetta-* and *yotta-* had to be invented. Although big data users tend to be close-lipped about the size of their installations, Uber, for example, boasted in 2018 of maintaining more than 100 petabytes of data to support more than 100,000 queries per day.

The rapidly falling cost of storage has several far-reaching effects on the ways organizations handle data. Instead of reducing data immediately to aggregate fields and throwing away the raw data, organizations tend to store everything they get. They are also willing to duplicate fields in different formats for different users.

Data engineering, as the name suggests, is a lot like software engineering. Data engineers learn to deal with such software engineering concepts as rapid and continuous development, automation and orchestration, modularity, and traceability. Operational concerns such as fault tolerance and service level agreements (SLAs) are handled differently in the age of data engineering. The appeal of clouds, both third-party (public) and on-premises (private), also alters the old equations while introducing new tools and processes.

A brief overview of jobs in data handling, and their evolution, might help set the context for the goals of this report. From the earliest work with data, housekeeping tasks were assigned to the DBA, who defined schemas, ran Data Definition Language (DDL) SQL commands, took care of backups, supported the data dictionary, and so on. Sitting between the DBA and the users, such as data scientists, data was prepared for use largely by a new position called the ETL (extract, transform, load) developer. But more modern tools stress application programming interfaces (APIs) and the preparation of data streams to supplement or replace the old DBA tools, leading to today's understanding of the data engineer. The traditional job of ETL developer is closest to modern data engineering.

For people who have learned DBA tools and practices, the transition to data engineering will be difficult. Software engineers also face tall hurdles in becoming data engineers. The responsibilities and tools are substantially different from those found in applications or systems programming.

This report traces the ripple effects of technical, social, and business evolution on data storage, data-processing tools, and data-handling procedures. It explains why data engineering has taken its current form and what skills you need to be a data engineer. This report is aimed at DBAs, software engineers and developers, students preparing for data-engineering careers, managers responsible for data, and others who need to understand how data can work in modern organizations. It covers the following:

- Major tasks of the data engineer
- Business intelligence and big data
- Different levels of structure in data and how to streamline data access
- Capabilities of third-party cloud options
- Tools for ingestion, transfer, and enrichment
- Managing and planning the infrastructure to run pipelines
- Software engineering development
- Automation and orchestration of data engineering

Data Engineering Today

Key issues that data engineers handle include performance, scalability, fault tolerance, change management, and exception handling. I look at these issues in this report and mention some of the popular tools available to solve them, along with the theoretical and practical knowledge you need. The next few sections show how the concepts and thought processes data engineers use resemble older ways of thinking about data, and discuss what assumptions need to change:

Data ingestion and transfer

In traditional environments with data warehouses and relational databases, this task is divided among a number of tools for different stages of data use. For instance, an SQL database might ingest data from an outside source, such as a spreadsheet, database, or flat file. To ingest data from an operational database to a data warehouse or a data store used by business intelligence (BI) tools, the developer applies a tool for Extract, Load and Transform (ELT) or the aforementioned ETL. Sometimes, replication or streaming tools are used to keep the target system up to date in near real time. Data virtualization tools can make data available on demand without having to move it and keep it up to date. Backups also use specific tools.

Big data environments divide tasks differently. SQL is used mostly for exploring the fields and characteristics of data sets (such as ranges) before production use, when it is supplanted or complemented by APIs. Backups probably require specialized tools—and are performed in cloud solutions through configuration options—but replication is built into most data stores and is configured as part of their setup. The new environments still use bulk data transfers and ETL tools, which have evolved with the times.

Transformations and enrichment

The data engineer often must add or change fields prior to storage. Reasons include:

- Correcting errors.
- Joining databases on fields they have in common, perhaps with changes to column names for consistency and clarity.
- Adding provenance metadata, such as the source of the data and a timestamp recording when the data was received.

- Converting different units of measurement to a common unit, such as English and metric standards, or different ways to represent values, such as "DE" and "Delaware."
- Precalculating new fields that users will need, such as the average for a column or a total price based on several different components of a price.
- Determining the column that holds data based on hints within the data, such as searching for a company name in a news article or social media post and storing the correct company name as a "company" field.
- Summarizing multiple rows to reduce storage, such as storing a few aggregate statistics on the data for an entire day at the end of that day. This is called *rolling up* a table.

The time and computing resources required for transformations and enrichment determine whether you perform them as you ingest the data or later, after the raw data is stored. Most sites maintain data in different stages: some closer to the raw state in which it arrived, others successively more refined. As the data becomes more refined, it comes closer and closer to meeting the needs of the organization's users. For instance, some users might receive only averages, without seeing the individual rows of data on which those averages are based. This reduction can be done for privacy reasons or just to reduce the cost of transferring and processing the data.

Cataloging

As data stores multiply, users have more trouble finding out what data supports their work and where it resides. To serve the users, data engineers must do more than ingest and store the data in an accessible format—they must catalog it so that users can search for it on relevant metadata. Such metadata could include column names, sources, date of creation or ingestion, keywords or other tags, and who owns the data within the organization. Catalogs are data stores themselves; the user queries the catalog, and then either queries the data store it points to or asks the data owner for access to that data store.

Storage

Data warehouses and relational databases are still important, but they have been joined by new types of data stores, including

temporary storage areas used for analytics. A diverse but well-organized (and well-cataloged) modern collection of data stores is called a *data lake*.

Like DBAs, data engineers are responsible for capacity planning. Cloud storage alters the calculations considerably because you can scale quickly and easily. But costs must always be kept in mind, because storage and network transmission are not cost-free. Furthermore, the tools used to ingest and process data in the cloud tend to be different from those you use on-premises.

In modern environments, data tends to grow quickly for many reasons: the sheer volume of available data, the constant emergence of new sources and types of data, the presence of data in many versions for experimentation, and the organization's desire to keep old data around because it might be useful for analytics and historical tracking. Thus, it's important to choose storage options and tools that can handle data growth in the range of one or two orders of magnitude.

Some organizations have entirely migrated away from one-size-fits-all relational databases. Others keep these databases for transactional applications while adding parallel data stores that contain data in formats more appropriate for capturing streaming data and analyzing big data. You will probably find yourself setting up several different types of data stores and copying data between them.

Partitioning, which is already common in the relational world, is even more crucial with big data because data sets are simply too large to store or process on a single host. This report looks at some of the considerations for partitioning. Recent data stores build in replication and partitioning, making fault tolerance easier.

Configuration tends to be simpler with the new data stores than it is for relational databases, but you trade flexibility for this simplicity. In other words, each of the new storage types is designed for particular uses of data. Your effort goes into determining the proper type of storage for each use, instead of tuning a single database to support that use. Retention requirements may also determine the choice of storage (see "Object and Tiered Storage" on page 33).

Automation and orchestration

Fast computers, comprehensive networking, and copious tools for automation have made it easier over the past couple of decades to let the system do boring and repetitive tasks. At the same time, every organization wants to do more work with fewer people, and you need to enhance your value to the organization by learning to automate. Besides, who doesn't want to be relieved of boring tasks? (Many programmers say that they would rather automate a task than perform it repeatedly, even if the total time required is the same.) When tested thoroughly, automated tasks are more reliable because they don't type commands incorrectly or forget steps.

Automation is a part of *orchestration*, or setting up processes that take your system through all the steps needed to perform routine tasks.

Security and access

In large environments, access to data may be the responsibility of other administrators, but some of the responsibility often resides with data engineers. Many of the tools in this report connect to authentication servers and offer Secure Sockets Layer (SSL)/Transport Layer Security (TLS) for data encryption in transit.

Access control lists (ACLs) can help to group users and control which groups have access to different types of data. Fields that are sensitive—because they contain corporate secrets or personally identifiable information (PII)—must be segregated from other data. You might need to learn new procedures for the new storage systems used for big data, and the data stores are integrated in various ways with the cloud's identity and access management (IAM) tools. (See "Limitations on Data Use" on page 12 for more about security and legal compliance requirements.)

Some changes brought by modern data environments are quantitative. In the age of the "knowledge worker," more users want access to their own data and the opportunity to create their own analytical tools. So the volume of requests for data can be greater than in old environments, and automating access will benefit you. Furthermore, the variety of data sources and stores makes protection more complex, and the volume of data stored makes breaches more serious.

You also need to structure data so that your users don't see sensitive data to which they are not entitled. Some modern data stores provide views, as relational databases do, so that you can present to each user just the rows and columns to which they have access rights. Because data duplication is common in modern data environments, you can create custom data sets for users.

Development life cycle
: All the trends discussed so far call for a clear set of steps in the evolution of your processes for handling data. The volume and variety of big data require repeatable, reliable processes. Security will be woefully lax if you deploy buggy software and processes, so strong software engineering practices and good orchestration tools can prevent problems. And opportunities for automation allow you to meet all these needs as well as force you to define the procedures carefully.

Few organizations assign all these tasks to a single person. Data engineering is normally a team effort, with senior members of the team setting policies and other members handling designated tasks.

The field of data engineering has started to get attention in the publishing world. One blog post lists books for data engineers to read, although not all of these books focus on data engineering. Some talk about data warehousing, which is different, or about tools shared by data science and data engineering.

Evaluation Process

The data revolution is driving, and is driven by, a new empowerment among employees at many organizational levels. This is called the *democratization of data* because organizations extend access to data to people below the management layers that used to have exclusive access. The field of BI is being transformed, like so many other organizational practices.

As a data engineer, you need to work closely with the potential users of your data to give them the data sets they need, in the structure they need. Questions to ask include:

What data sets do users need?
: This helps you prioritize which to process and ensure that they can be found in catalogs.

What fields do users query, and what data do they want in return?
: Answers to these questions help you structure data for fast retrieval.

How fast do they need data and at what granularity? Do they want real-time access to particular transactions, or are they satisfied by weekly summaries?
: These questions help you choose the processing tools and storage for different types of data, determine how many pipelines to use for the performance you need, and create useful aggregate data.

What data is sensitive and needs to be protected from unauthorized employees?
: The answers can affect the structure of both data and access groups.

Ask users to try to think two or three years ahead so that you can design systems to scale and be adaptable. But because anticipating needs in modern business environments is difficult, the processes described in this report allow you to create new forms of access to data quickly.

Organizations that fail to collect these requirements early in the data engineering effort—or that fail to repeat the collection regularly—end up with extremely sad outcomes. Data goes unused and employees express constant frustration at the delayed or missing data they want. Angry managers might stop analytical projects and write off their investments in the related tools and employees as a loss, even though the problem was never with the tools or the goals—it was the failure to meet the users' real, specific requirements.

Many departments hear about enticing new software (Hadoop! Spark!) and invest in it to appear up to date, without doing enough research about what benefits the tools actually provide. If these tools end up being inappropriate for the organization's data types and uses, the tools will simply waste resources (including the learning time spent by employees) and lead managers to question the goals of data-driven transformation.

Best Practice

Never put tools or data in place without a detailed plan for how you will use them.

Data engineers may know better than other employees whether a particular data set is fit for certain purposes. Data possesses a huge variety of aspects, each of which can influence its value for some particular type of analysis. For instance, a data set may look impressive because it covers 200,000 people, but if that data contains very few representatives of some population (say, Native Americans), it might not produce accurate results—at least when applied to Native Americans. Concerns with aspects of data are common in many fields. Statisticians, for example, learn early in their studies that they need to choose the right test for their data; for instance, the popular T-Test is inappropriate for nonlinear data. Machine learning introduces all sorts of other considerations. Thus, data engineers as well as data scientists often run exploratory queries on new data sets to learn the characteristics that will matter during analytics.

As the person who may best understand the available data, the data engineer must speak up and help users choose the right data and the right kind of analytics. You might point out that the window during which streaming data is captured must be large enough to provide enough information to produce a meaningful summary—or that a window must be made smaller so that important anomalies don't get lost when aggregating them into a large data set.

If a business user asks you to use streaming processors on log data that is stored for archival purposes, you might point out that a simple write to a file would suffice. As another example, data used for legal compliance must be checked for adequate accuracy. And there are plenty of applications where traditional SQL queries to a relational database are more appropriate than the fancy modern tools that get a lot of press.

Data prep also depends on the shape of the data and how it will be used. For instance, you will probably handle sensitive data that needs to be anonymized in order to prevent misuses, such as reidentifying individuals whose identities must remain hidden. You often have to find a balance between opposing risks: reidentifying individuals versus making data so general that it loses its value. Statistical analysis might tell you, for instance, that in certain high-population

areas you can record the city in which a person lives, whereas in lower-population areas you should scrub the city out of the data and record only the county. In food stores, a record of a $15,000 purchase is outrageous and should be flagged for potential errors, whereas at an automobile dealership such a purchase is on the low side.

If the shape of data changes, you must help business users reevaluate its suitability for their applications. Organizational needs change also. For instance, a speed-up in decision making or a push for finer-grained accuracy may call for more data and faster processing.

Business Intelligence (BI) and Serving the Analysts

Before the pressures of modern data processing led to constant, changing requirements for analytics, a kind of staged, waterfall approach to developing data was in effect. The analysts would think up a research project and ask for a graph or table of data. The DBA would find the data and provide it to programmers who would create the visualization for the analysts.

In today's data engineering environment, analysts want immediate access to data, although they will probably wait for the data engineer to clean it. The analysts will then create new tables of derived data, which they want updated quickly—perhaps on a real-time basis—with the latest data. So the data engineer will implement the analysts' transformations as a pipeline that accepts streaming or batch input and outputs the necessary visualizations. One way to look at the relationship between analysts and data engineers is that analysts create a prototype, after which data engineers create a production system.

Example of Data Exploration

Qubole used this kind of iterative process to create a Presto-as-a-service connector for Microsoft's business analytics tool Power BI, designed to help data engineers and data scientists uncover important fields and their relationships by running queries across multiple data sources. Data analysis often combines fields from relational and nonrelational data. You can read about it in more detail on the Qubole blog.

Batch versus streaming

Data engineers serve many types of users in different ways. Although streaming data, which comes in quickly and continuously, is the hot topic in data circles, most data analysts still derive insights from large quantities of data through batch processing. When this report contrasts batch data with streaming data, it's not talking about some trait inherent in the data, but simply how the data is processed. If you store it in a file or database and run analytics over large numbers of records or rows relatively slowly, it's batch data. The very same data may be delivered in a steady stream and consumed one message at a time. Thus, streaming data tools also often operate on batches of data sent as files. Sales information and other transactions often come through this way.

Many applications treat the same data in a batch and streaming manner. For instance, a fraud detection application might run analytics over large data sets to assign traits to customers, then compare those traits to the same data received in real time in a streaming manner to determine whether a particular credit card transaction is fraudulent.

"Streaming Data Processing" on page 41 lays out some popular tools for streaming data and their uses in data engineering.

Limitations on Data Use

With the rampant collection and crunching of data, ethical issues also arise. Governments, as well as the general public, are now taking security and privacy more seriously, as we see in the rush to prevent a repeat of the kind of use Cambridge Analytica made of Facebook data. More recently, Twitter has admitted to misusing data for advertising. Everybody understands the need for Twitter to collect personal data to serve its users, and Twitter advertising is also widely accepted—the problem comes when data is used for a purpose that it shouldn't be used for.

As another example of the importance of social expectations, consider the famous case in which Target sent pregnancy-related offers to a 17-year-old who was trying to hide her pregnancy from her father. This public relations disaster highlights the differences between laws, ethics, and plain good sense about business goals. Legally, Target was perfectly entitled to send pregnancy-related offers. Although there's a difference between personal medical infor-

mation and routine retail sales like potato chips, Target was also probably within reasonable ethical bounds in sending the pregnancy information. Where the company fell short was in considering what customers or the general public would find acceptable.

These social expectations also vary based on the type of data: most people are much more worried about the sharing of medical or financial data than data about everyday purchases or location data, although anything is potentially open to abuse. Even everyday locational data or postings of photos with labels to identify people can be a life-and-death matter for victims of domestic violence who are hiding from abusers.

One practice adopted by many organizations is to flag sensitive data through metadata; you might mark data with a pointer to the terms and conditions under which it was collected. Another good practice is to store sensitive data in completely separate data stores from less risky data. No one should even receive the data within your organization if they are performing tasks that weren't included in the terms and conditions (as in the Twitter case just cited). Finally, you can de-identify data if the precise values aren't needed for certain types of analytics and for duplicate versions of data used for testing. De-identification may involve substituting synthetic data for real values or switching around fields, so that the totals are all correct but the values (such as age or medical diagnosis) are assigned to people at random.

Higher stakes are imposed on improper data use by legal requirements such as the European General Data Protection Regulation (GDPR), the US Health Insurance Portability and Accountability Act (HIPAA), and the CAN-SPAM Act. Violating such regulations can lead to millions of dollars in fines, and the European Union in particular has shown the resolve to take the GDPR seriously. So it's important to stay up to date on standards and regulations affecting data use in your industry, as well as to set up your systems to avoid inadvertent violations by your users.

One headache added—probably unintentionally—by privacy laws to the data engineer's task is the "right to be forgotten." If someone living in a jurisdiction covered by a "right to be forgotten" law (includ-

1 Disclaimer: this paragraph is an extremely simplified summary, not legal or technical advice.

ing the European Union and California) presents a court order telling you to suppress data about the person, or if consent to store the data is withdrawn or expires for other reasons, you have to stop sharing data about the person, at least within that jurisdiction.

You might comply with a "right to be forgotten" order by finding every instance of the data and permanently removing it, or keeping it for internal use and aggregate data but making it unavailable within that jurisdiction. The data you need to delete might already be in many different columns in a number of tables and data stores. Deleting data could also make the data store inconsistent with previously calculated aggregate data, such as an average, although the effect of a single deletion is probably negligible.

Some sites maintain PII in one place and link it to a random identifier that contains the non-identifying information. If instructed to forget the person, these sites can simply delete the link between the PII and the random identifier, anonymizing the person's information.

Beyond the firm conditions imposed by terms of use and regulations, it is valuable to think about the purpose of data use. Will the uses open up new opportunities for your employees and clients? Or will they exploit people and put them at a greater disadvantage in relation to large institutions that control aspects of their lives? Constraining people from doing bad things, such as committing fraud or posting false news stories, is necessary in order to give legitimate activities room to spread, but most data use should be aimed at supporting people in doing what they desire.

Data Is Different Today

When computer databases first became widespread, most data came from human input: paper forms sent in by customers, receipts from sales, and so forth. Nowadays we have a plethora of data sources: sensors, cameras, log files from web servers or other hubs, social media, and more. Naturally, the volume of data is much greater. Other changes in data are more subtle.

Architectural challenges

The types of errors generated by automated input are different from errors in human input. Error-checking must be done differently, and the enormous data sizes call for automating this error-checking.

Provenance also becomes important: who supplied the data, what form it came in, and how trustworthy it is.

Errors in human-entered data include:

- Simple typographical errors, such as typing "Carloine" for "Caroline"
- Data typed in the wrong field, such as entering the state in the field meant for the city
- Incomplete records
- Skipped records, records entered twice, or data from one record mixed with data from another

In contrast, when data is taken from machines or automatically generated sources, errors tend to be more systemic:

- Corrupt files that can't be read
- Columns that are marked incorrectly
- Missing metadata
- Values that are way out of range, such as temperatures of 600,000 degrees for a factory machine
- Values that are converted incorrectly, such as making every field 100 times the value it's supposed to have or (famously) interpreting measures as metric units when they were provided in imperial units

Many errors that would be caught through traditional ETL, because they break the schema, can slip through in the modern data era. For instance, if the person doing data entry put a name into a field for age, a database would reject the record because the age has to be an integer in order to be stored in that column. A representation in JSON has no built-in check to make sure the age field is an integer. Some modern data formats such as Parquet would reject the bad field, because these formats define data types similar to those found in relational databases.

Automation is required to catch both human-entered and machine-entered data on a large scale. For instance, commercial tools exist to read large columns and determine that "DF" is not a correct state abbreviation, but that "DE" appears many times and might be the right substitute. The tools can autocorrect where changes are obvi-

ous and present unclear errors to a human for correction. Data prep may take place in several stages, and some of the advanced processing might not be done until users make a specific request for data for a particular purpose.

You should understand the shape, quality, and granularity of your data in order to help business users use the right data and choose appropriate analytics, as discussed in "Evaluation Process" on page 8. Provenance metadata can also specify these aspects of data that allow it to be used accurately and productively.

Architectural evolution

All the practices discussed in this report spring from a fundamental change in the structure and storage of data. For three decades, the notion of "Don't Repeat Yourself" (DRY) or "a single source of truth" drove relational design, and in particular the concept of normalization. When E.F. Codd published his historic paper on the relational model in 1970, expensive disk storage also mandated eliminating duplication.

The classic relational model forced a single source of data to satisfy all needs. Indexes sped up access to various columns so that one user could search by date and another by customer name. Now the factors of the storage equation have changed: we prioritize fast access at the expense of added storage. Opening multiple tables to get, for instance, a purchase order followed by customer name followed by address would be too slow. Better to keep all the information you need in a single row, even if this means repeating yourself.

As the size of data swelled and performance declined during the 1990s, the classic relational model was relaxed by providing summary tables and other conveniences in data warehouses. But by the early 2000s, it was time for a new paradigm. The solutions that were discovered during that period speed up access to data using a variety of practices. Most of these practices prioritize read performance over write performance, and therefore work best on data that is written once or updated rarely.

2 Originally in Volume 13, No. 6 of the Communications of the ACM.

Best Practices

- Data meets different needs by being duplicated in different tables and databases, each sorted by a single key instead of providing multiple indexes. Because retrieval from such dedicated data stores has become simpler, users tend to interact with them through programming languages instead of SQL.
- Data is partitioned by key to allow each user to query a single node, or just a few nodes, to get the data needed for the application such as the sales for a single country. Access is accelerated even more by reading a whole data set into memory, when possible.
- Partitions, whether done dynamically on streaming data or statically on stored data, allow parallel processing by short-lived processes that live in virtual machines or containers.

Because the data stores discussed in this report are used for analytics instead of transactions, users don't ask for strict consistency, such as Atomic, Consistent, Isolated and Durable (ACID) guarantees. Data is processed in stages to provide better value to users, with the goal being to get fresh data to the users quickly. The complicated acknowledgements and checkpoints that would be needed to ensure that every copy is in sync would just slow down the process. However, to preserve historical accuracy, the raw data is usually preserved as long as space restrictions make it practical to do so.

Structuring Data

Modern data stores categorize data quite differently from relational databases. The basics of relational data are:

- The database stores each item of data as a record or row, containing a fixed set of fields or columns. Columns cannot be repeated within a row. For instance, if a customer makes two purchases, you create a separate table of purchases and use a join table to indicate which customer made which purchases.

- Each column is marked with certain metadata, especially a type (such as *integer* or *date*) and a size. Many types, such as integers and strings, reflect the underlying representation of the data in the hardware and operating system. Getting column sizes right is important for efficient storage, and variable sizes are used only when fixed sizes are unfeasible, because more space is required to indicate variable-sized columns.
- Keys are important, and a unique key (one whose value has to be different in each row) almost always appears in each table.
- Relational data cannot be nested. Most relational databases nowadays allow you to store certain structured formats such as JSON or XML, but they are a foreign format embedded into a relational database column. If you query for a JSON field, you are actually running a JSON engine inside the relational database engine.

The preceding rules feed into the rules of normalization, which are tied up with the relational model. Most sites find it useful to violate the rules of normalization to improve the performance of certain operations. Providing data in different tables for different users is a common feature of data warehouses, and it is at least as common in the age of big data. You must be willing to convert the data into a different schema or group of partitions for different applications. Applications that process millions of rows can't afford to query six tables to retrieve all the data they need; instead, a data engineer prepares a table in advance that combines all the data needed by that application, and perhaps performs some preprocessing to store aggregates such as maximums and averages.

Many of the traits of relational databases change or become irrelevant in other types of data stores that are now broadly popular. Some superficial traits remain: big data still represents items of data as rows, and the columns or fields are often marked with data types such as integer and date. Keys are also nearly universal. But most other common rules from relational data disappear, and normalization is ignored or treated as an option for certain appropriate subsets of data.

Many modern data stores allow arbitrary fields in each row, as in JSON or XML. There can be fields in some rows that don't appear in others. And there can be multiple fields of the same name in a single row, so that two or more purchases could appear in a single cus-

tomer row. If you choose to create a separate table of purchases, that's probably not done to normalize the data but to speed up operations that search for and manipulate purchases. Nested data structures are common in data rows, along with data structures not recognized in the relational model, such as lists, arrays, and maps.

Because storage is more ample and compute speeds are faster than they were in the 1970s when the relational model was designed, variable size is commonplace and few data stores ask you to assign a size to a column.

Fields, Columns, and Schemas

The key difference between the relational model and big data projects is that the latter have a looser idea of a schema.

Traditional projects involve long planning times to define relational schemas before data collection can begin. Even though big data projects are often called "schema on read" (which suggests that the data can be written in any format and structured later by the user), the projects need planning for data structures, too—but this planning requires significantly different kinds of thinking.

MongoDB, Cassandra, and other nonrelational data stores are by no means raw data; they still expect input to have structure. But the structure can differ for each row. These kinds of databases are sometimes called document stores because they have nested key/value fields resembling the structures found in HTML or XML documents.

If data arrives in a key/value structure such as JSON, storage is easy. But unstructured data, such as postings on social media or multimedia files, requires some work to add identifying metadata. For instance, you can add a date-and-time stamp to serve as a key.

The MapReduce programming model used by Hadoop expects each record to start with a key. The rest of the record is the value. It's the job of the programmer to create each key. Thus, if a MapReduce program is seeking to collect data on different countries, it might extract the "country" field from its input and use that as the key. The key will also start the record when it is stored.

Best Practice

If you know that a field will be specified often in queries, consider adding an index to it.

Most modern data formats also support indexes, which serve queries in ways similar to relational databases. In addition to support by databases, indexes appear in several formats that are popular for storing big data. It's optimal to add these indexes after data has been loaded so that data is smaller during initial ingestion.

Example: Duplication and Normalization

"Architectural evolution" on page 16 explained how the current generation of data maintainers has accepted the duplication of data. Let's look now at a small example of structured data to see why you need the flexibility that modern data formats offer instead of sticking to the relational model. Imagine a simplified data structure for a purchase, as shown in Figure 1-1.

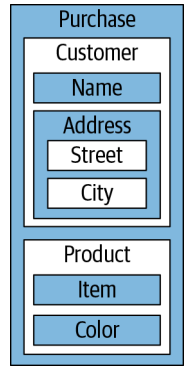

Figure 1. Sample record structure

A JSON representation of a purchase might look like this:

```
{
  "Purchase": {
    "Customer": {
      "Name": "Alan Fentolo",
      "Address": {
        "Street": "56 Spring Road",
        "City": "Jonesville"
      }
    },
    "Product": {
      "Item": "Tie",
      "Color": "Green"
    }
  }
}
```

Suppose an application retrieves all the city fields from several million purchases to improve marketing. In each purchase, the application has to find the customer field. Then, within the customer field, it has to find the address field, and within that find the city field. This means three retrievals for each city, a task that a programmer would call *dereferencing* fields. If structures are deeply nested, the extra dereferencing could add up and degrade performance. (The problems of relational databases are similar because a search for a field may require joining tables or querying one table to get an ID that you search for in another table.)

So you might choose to take the city field out of the customer field and make the city a top-level field within the purchase. This saves two dereferences.

And what if another application retrieves item fields from the product fields? A purchase can probably contain several products, and you might want to know with which product each item is associated. So you might keep the item nested within the product or create a new field that combines the product and item.

Best Practice

When you duplicate fields, to maintain a single source of truth, use metadata to mark the original copy that becomes the source for other copies. Enforce a data flow that updates the original copy and then lets the update flow out to other copies.

In the old way of working, a DBA would give users access to selected fields and rows through views, avoiding the need to copy data to a new database. Views are also supported by some of the newer database engines, such as MongoDB, and can be useful in data engineering.

Structured Storage Formats

Some data stores have their own internal formats. But many big data projects need input or output in a standard, interchangeable format. The Hadoop Distributed File System (HDFS) also allows data to have any structure of your choice. So data engineers can spend considerable time researching different storage formats and choosing the right ones for their applications.

One major choice is whether to store data row by row or column by column. Traditional databases store data row by row, with all the columns in a row stored together (except for a few particular cases, such as large unstructured fields known as binary large objects [BLOBs]). This storage makes it easier to write data because most operations add or update a set of rows through a WHERE clause (for instance, WHERE COUNTRY = US). These databases are used mostly for transactional applications, which retrieve several columns from a particular row for a customer, a product, or something similar. Row storage here is natural and convenient because the data wanted by each query would probably be stored together on disk.

Many modern applications perform much better with columnar storage because column sizes can vary more widely. Therefore, more

recent database offerings—relational and nonrelational—often feature columnar storage. Relational databases tend to offer multiple options for ordering data, and let you run a single query over these different storage formats.

Avro, ORC, and Parquet are popular storage formats in the big data world. Avro uses row storage, whereas ORC and Parquet use columnar storage. In other ways, they provide very similar functionality. They let you create hierarchical data structures, support lists or arrays, store the data in binary format, and offer various types of compression. ORC and Parquet offer indexes.

Typically, you will send and receive data in JSON format. This format was developed for use with JavaScript (hence the J in the name), has become the most popular format for data exchange on the web, and is turning up in all sorts of other contexts requiring structured text data. For internal storage and processing, you will convert the data to one of the binary formats such as ORC.

Best Practice

The cost savings you get by transferring and storing binary data will be well worth the effort of conversion back and forth between text and binary formats.

Naturally, there are differences between the formats. Avro is key/value, whereas ORC and Parquet have primitive types such as in a programming language: numerical, string, date, time, and so forth. You can build up structures using the primitive types, and Parquet also allows you to create logical types on top of the primitive ones.

There are other storage formats for structured data, such as Google's protocol buffers and Apache Thrift, developed by Facebook, but the most popular formats currently are Avro, ORC, and Parquet.

Although these formats do not fit into relational schemas—because they are hierarchical, offer data types such as lists and maps that SQL doesn't understand, and are incompatible in other ways—they can be read and written using the SQL technologies associated with big data, such as Hive, Impala, and Presto. Major programming languages also provide libraries to work with the formats, so that applications can easily read and write them.

Data Warehouses and Data Lakes

Modern data stores were developed in the 1990s and 2000s, when sites handling big data found relational databases too slow for their needs. One major performance hit in relational databases comes from the separation of data into different tables, as is dictated by normalization. For instance, a typical query might open one table and offer a purchase number in order to obtain the ID for a customer, which it then submits to another table to get an address. Opening all these tables, with each query reading new data into the cache, can drag down an analytics application that consults millions of records.

To accommodate applications that need to be responsive, many organizations created extra tables that duplicated data, stretching the notion of a single source of truth to gain some performance. This task used to provide a central reason for ETL tools.

The most popular model for building storage of this type as a data warehouse involves a "star" or "snowflake" schema. The original numeric data is generally aggregated at the center, and different attribute fields are copied into new tables so that analytics applications can quickly retrieve all the necessary data about a customer, a retail store, and so on. These non-numeric attribute fields are called *dimensions*.

While traditional normalized databases remain necessary for handling sales and other transactions, data warehouses serve analytics. Hence the distinction between online transaction processing (OLTP) and online analytical processing (OLAP). OLAP can tolerate the duplication done in a data warehouse because analytics depend on batch processing and therefore require less frequent updates—perhaps once a day—than OLTP.

But in modern business settings, such delays in getting data would put organizations at a great competitive disadvantage. They need to accept and process streaming data quickly in order to reflect the failure of a machine out in the field or to detect and prevent the fraudulent use of a credit card. Data must also be structured on the fly; data analysts can't wait months or years for a DBA to create a new schema for new data.

Therefore, the structure of relational databases, which handle analytical processing in multiple steps, renders them inappropriate for

big data. The new data stores look back in computing history to databases with simpler structures that don't try to be all things to all people. This means you have to choose the particular data store that's best for each application, then determine how to structure the data to make access as fast as possible.

Modern data environments tie together data sets of many types and sizes, refreshing them from multiple data sources. Because these collections of data are handled so differently from data warehouses, organizations like to call them *data lakes*. Often, organizations also link to outside sources and retrieve data from them as needed. For instance, Server SQL's PolyBase lets you join multiple sources in an SQL query: internal and external, or relational and nonrelational.

Provenance and catalogs are key to making a data lake work. If you lose track of what you have in this diverse collection, you end up with what data engineers fearfully call a *data swamp*.

Database Options

The major categories of data stores are:

Relational
: The traditional databases developed in the 1980s and 1990s: Oracle, IBM DB2, MySQL, PostgreSQL, and so on.

Document
: These store data as labeled fields, like XML or JSON. Each record or row can be of arbitrary length. Sometimes these stores follow schemas, such as Avro, ORC, and Parquet. Other document stores, such as MongoDB and CouchDB, allow each record to have a unique structure, naming each field.

Key/value
: Stores each element as a key—which may or may not have to be unique—and a value. These databases hash the key to make retrieval by key fast, so key/value stores are sometimes called *hash tables*.

  The distinction between document stores and key/value stores is fuzzy. For instance, Cassandra started out as a key/value store, but has evolved into more of a document store. Any key/value store can store a document as the value.

Graph
: These represent relationships, such as networks of people or genealogies. For instance, each person can be a node or vertex, while the relationship between two people is represented as an edge. Graph databases are used by very few applications, but can be valuable for those applications, so their popularity is growing.

Search
: These are specialized document stores whose indexes offer sophisticated search capabilities. Examples include Lucene and ElasticSearch.

Best Practice

Always consider the evolution of your data when choosing a data store. Unless you are sure you are storing legacy data that will not need new columns or other changes, make sure to use a data store that evolves efficiently.

Comma-separated value (CSV) files provide an example of a rigid data store that does not evolve well. Suppose your data comes as CSV files with the three fields: `Repair type`, `Date`, and `Cost`. You might write an application that extracts the third field and stores it in a database as the cost. But then the organization producing the data could add a `Time` field after the `Date` field. Now your application is broken, because the third field is no longer the cost.

Hive is much richer and more flexible than CSV, but it actually suffers from similar problems. It generally uses HDFS as its data store (although other formats are available), and HDFS was designed to support individual applications. HDFS is not easy to adapt to future uses. So early releases of Hive didn't even offer an `ALTER TABLE` statement. Various `ALTER` statements, similar to those used in relational databases, were added later, but they are so inefficient that many experts recommend you just read all your data into a new table and overwrite the old one. Repartitioning can also encounter this performance problem.

A few factors determine the performance and reliability of the database engine used:

Locality
: If you can get all the data you want from a single file, access is much faster. To handle big data, which rarely fits onto a single system, data is partitioned into multiple files.

  Furthermore, within the file, data that is sequential can be read much faster. The query manager can get large chunks in a single read, can take advantage of sequential access to read ahead, and can often retrieve the following values from the cache instead of returning to the original file. This aspect of locality leads to the choice between row storage and columnar storage.

Metadata
: Variable-length character data takes up more space and takes longer to query than fixed-length character data, because you need extra metadata in each row to record how long each variable-length field is. Similarly, document stores take up more space than database schemas because each record can contain arbitrary fields. Thus, the key for each field must be stored with the value. Some modern binary data formats (notably Avro) optimize this extra storage by defining a schema and applying it to the row.

Locks
: All databases used in big data allow concurrent access for performance reasons. To prevent corruption, rows or tables must be locked during reads. These locks take up space, and processes trying to write data must wait for the locks to be released. Various schemes work around locking, such as by assigning versions to data as it is updated, in order to minimize the performance degradation caused by parallel access during writes.

Each database makes different trade-offs for these technical choices. The underlying files used by a database on its host operating system are normally hidden, but some aspects may be exposed; for instance, the administrator may have to allocate space for these files and back up the files directly, instead of issuing commands at the database level. Some modern databases, such as Hive, let you choose the kind of file storage they use. Most people use Hive on top of HDFS, but you can use flat files or other types of storage.

Access to Data Stores: SQL and APIs

Hadoop and other modern, nonrelational databases provide APIs, which can quickly carry out the basic database operations known as CRUD (Create, Read, Update, and Delete). Thus, it's well worth learning a programming language in order to use these APIs. SQL, even though it's the common way people interact with relational databases, should be considered a high-level language that imposes overhead. The SQL query optimizer takes up overhead through such activities as deciding on the order in which to execute a WHERE clause, and whether to consult indexes. Thus, even relational databases have raw, direct APIs. Someone planning an application and exploring the data will find SQL useful at that stage, but most production applications in big data use APIs.

The Hadoop family of tools was designed in the mid-2000s and mostly offers Java interfaces. But many people find it easier to use Python, Scala, or other interpreted languages. Because you can try out interpreted language statements using an interactive command line, development can go faster. These languages also lend themselves to more compact source code than older languages such as Java. If you learn Python or Scala, you can probably interact with any of the modern databases. Jupyter Notebooks, which are interactive environments allowing you to try out programming code, are also popular to start development.

But because both developers and DBAs have been accustomed to using SQL, and because it can be a convenient way to explore a data set, nearly every database supports some SQL-like language, even if the database is a document store or has some other kind of nonrelational structure.

HDFS is just a filesystem designed to handle large files with replication. The filesystem imposes no structure on the data within it, but Hadoop and most other programs use it to store individual records with keys. Hive and Impala create schemas on top of HDFS to replicate the relational model. You cannot use these extensively to execute complicated queries, though, because the underlying data stores don't have the relational structure that supports the execution of arbitrary queries. If you try to include too many clauses, performance will be unfeasibly slow. For instance, the early versions of Hive didn't even have UPDATE and DELETE statements, because the underlying HDFS filesystem is optimized for continuous writes.

Hive and Spark SQL are frequently used SQL interfaces to HDFS. Another interface, Pig, implements less of the standard SQL language, but offers special operations that are useful for filtering and other big data operations. Pig balances traditional SQL queries with the kinds of programming functions used for analytics.

Best Practice

For big data, engineers should get used to programming APIs for ingestion, transformations, and other production purposes. SQL is useful to explore data sets, and as an interface to data catalogs so that users can discover data sets.

Hare are some newer SQL-based tools that are attracting attention:

Iceberg
: Iceberg claims to solve some performance and reliability problems found in HDFS. Even though it's still in the "incubator" stage, it's gaining a lot of attention. One of its goals is to allow in-place table maintenance. A schema redefinition, such as ALTER TABLE, will work much faster in Iceberg than in Hive.

  Iceberg partitioning is more sophisticated than Hive partitioning. When you query data, Hive requires you to specify which partitions to search, along with the values in a WHERE clause. Iceberg has a feature called *hidden partitioning*, which infers the partition containing the data from your query. Thus, if you have partitioned the data by a date column and your WHERE clause specifies that date column, Iceberg will automatically find the right partition.

  Finally, Iceberg handles writes in ways that the developers claim will benefit both performance and reliability. For instance, it supports concurrent writes by creating a new file and performing an atomic swap if there is no conflict. This is a typical computing tradeoff: using more memory to save time.

Presto
: Presto may be replacing Hive and Impala as the preferred SQL interface to HDFS. It's appealing because it also works with MongoDB, Cassandra, ElasticSearch, Tableau, traditional databases, and even flat files; it has about 20 connectors to such databases as of early 2020. It can also work over Kafka to handle

streaming data so that a single Presto query can combine data from multiple sources, allowing for analytics across your entire organization.

Presto is designed for distributed queries, working across many nodes in parallel. You can tune it to respect the amount of memory on nodes where its workers run, and you can set limits on resources used. You can combine nodes into *resource groups* and allocate different amounts of resources to queries on these resource groups.

Authentication uses Kerberos, but a simple form of username/password authentication with Lightweight Directory Access Protocol (LDAP) is also available.

SQL has met critical research needs for decades, so it can still be valuable for issuing the kinds of queries where it functions well on both relational and nonrelational databases. Recognize, however, where other types of batch and streaming analytics work better with the newer tools and APIs.

Cloud Storage

More than three-quarters of organizations now use third-party cloud services, which offer a variety of free/open source and commercial data stores along with their own proprietary offerings. Some vendors claim that their proprietary data stores offer better performance than other data stores running in the cloud. Nevertheless, many data engineers avoid the vendor's unique offerings because they want data to cross cloud boundaries. They might be running a data store on-premises as well as in the cloud—a hybrid public/private deployment—or they might simply be afraid of vendor lock-in. You can easily transfer data in and out of the cloud, but your use of the data may become dependent on the optimizations offered by your cloud vendor.

How cloud storage differs and is similar to on-premises storage

Whether you deploy a standard data store or one of the cloud vendor's alternatives, the advantages in scalability and reduced administration certainly make the cloud appealing. Security is probably controlled better in cloud environments than the staff in most organizations can muster, although the use of the cloud has no impact on the most common reasons for breaches: abuse by insid-

ers, and access to employee keys by intruders through phishing or other attacks.

When you run your own data center, you can design it to keep related items of data close together, thus improving performance. For instance, all the partitions needed by a particular application could be stored on a single rack. There are similar capabilities in the cloud: besides choosing a region and availability zone close to the user, you can configure the cloud service to locate nodes in proximity to one another.

As you evaluate major cloud offerings, you will encounter unfamiliar tools developed by each vendor, along with familiar offerings running inside the cloud. This is a bit like visiting a grocery store in a foreign country: popular products in one country are often copied and offered under a new brand name in another. However, products from your own country may also appear under their own brands. The same goes for tools in the cloud.

The major vendors compete vigorously, but each does things differently. So you might find that something you need is very simple at one vendor, but requires a difficult multistage process at another. Of course, each vendor will try to sell you their own solution as the best. It's a bit like buying an automobile: a luxury car dealer is very good at persuading you that you need their luxury car, but you might end up just as happy with a budget sedan.

Naturally, data engineering requirements are not the only considerations that go into choosing the cloud—many other departments will insist on various needs. But data engineering is the foundation for making data accessible and useful, so management should respect data engineering's requirements.

As in many situations where public or open source tools coexist with commercial offerings, you have to make a tradeoff between the power and convenience that cloud vendors offer and the risk of lock-in. The cloud offers ways to help you keep your options open, through tools that aid in data transfer (so you can maintain multiple cloud services or a hybrid system using your on-premises equipment together with the cloud) and standards in key areas such as data formats.

Platform as a Service and serverless options

An important trend in cloud computing is Platform as a Service (PaaS), or "serverless" computing, where you write functions and submit them to the cloud to run. PaaS is well suited to the trend in software development toward using containers. Developers using PaaS or serverless computing don't have to package an operating system with their functions, and they worry less about the operating system in general.

The major cloud vendors—Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP)—keep expanding their offerings month by month, with the goal of providing you with a complete environment. Like shopping malls, they hope to meet all your needs and keep you on their virtual premises all day. They also have moved "up the stack" to offer not just storage and compute power, but packages of tools that are easy to get started with.

Best Practice

You should be clear about how you will use data (see "Evaluation Process" on page 8) before talking to vendors and investigating their offerings.

Example options

Here are a few examples of tools from major cloud vendors that resemble the free and open source ones described earlier in this report:

AWS

Athena is an SQL query tool based on Presto.

Data Pipeline is an orchestration tool that can transfer or analyze data on a schedule.

DynamoDb is a data store that offers both key/value and document formats, as Cassandra or MongoDB do.

Glue is an ETL tool that can discover schemas in semi-structured data and ingest it into AWS data stores.

Kinesis is an ingestion tool similar to Kafka, supporting streams in multiple types of media.

Azure

Data Catalog helps you extract metadata and set up a catalog for users to search for data they can use.

Event Hubs is a streaming data ingestion service like Kafka.

Table Storage is a semi-structured database.

GCP

BigTable is a highly scalable key/value data store storing semi-structured data.

Dataflow is a framework for transforming and enriching both stream and batch data.

In addition to these three major vendors, Databricks deserves a mention for its popular analytical solution, built on Spark.

Object and Tiered Storage

The rethinking of storage among big data users has extended to the architectural depths of disk structures and operating system choices. As an alternative to the standard block storage offered by conventional filesystems, cloud services and open source projects offer *object storage*, meant for types of data that aren't expected to change or be edited. Object storage is popular among organizations that store large amounts of multimedia files such as audio and video, or that archive large amounts of data that they might need access to in the future.

Object stores also scale efficiently, making them a good choice when you want to quickly append BLOBs or store large write-once data such as logs. In the cloud, common object stores include Amazon Simple Storage Service (S3), Azure Blob, and Google Cloud Storage. Cloud services also offer tiered storage, where you can trade off cost for access time, and even define policies, so that (for instance) an object moves to a slower, cheaper tier after 30 days, then switches into a cold archive after one year, and finally is deleted after a time of your choosing to meet regulatory or tax requirements. Options like this appear on AWS, Azure, and GCP.

Object stores are cheaper than block storage because they don't provide random access to data. In fact, they don't even provide a directory structure, although sometimes they let you simulate one, just so you can keep a logical inventory of what you have.

Best Practice

Maintaining robust metadata on the objects you store is crucial, so that users can find objects later.

Object stores require study because they differ a lot from one another, as well as from block storage. You may have to learn a special API to read and write to and from each object store. Investigate the architecture of the object stores that interest you to understand how to best make use of traits such as scalability.

Metrics such as bytes that are added and deleted can help you make better use of the object store. For instance, metrics may let you know that it's not such a good idea to archive data after a year, because people are reading and writing it more often than you anticipated. "Metrics and Evaluation" on page 44 discusses the types of metrics you can capture and how they might be valuable.

Partitioning

Big data works by partitioning, or *sharding*, data. This lets distributed systems store huge quantities of data on different servers as well as process the data by dividing it up by natural sections.

Column selection

Choosing the right field on which to define partitions, along with choosing the right keys and indexes, is crucial for efficient data processing. Think of slicing a grapefruit: if you do it properly, you can easily extract the pulp, but if you slice the fruit at an odd angle, all the pulp is stuck in hard-to-access places.

A similar concept turns up in Kafka, a popular message broker, as *topics*. Publishers assign a topic, which is simply a keyword, to each record they submit to Kafka. Thus, a stock reporting tool might use the stock symbol of the company as a topic (MSFT for Microsoft Corporation, for instance). Consumers subscribe to topics in order to get just the records of interest to them.

Criteria for choosing keys and partitions

As one classic example of key/value pairs for big data, let's look at the paper introducing MapReduce by Jeffrey Dean and Sanjay Ghemawat of Google. Google needed to create a list of web pages from

around the internet, indexed by each word. For instance, this report would appear with "report" as an index, then again with "appear" as an index—one appearance for each important word in this sentence. Thus, their MapReduce algorithm accepted input records with a word as the key and a URL as the value.

When you partition data for storage on a cluster of nodes, the choice of key you use for partitioning depends on how applications will use the data. One of the most important impacts on performance is the number of files the application has to interact with. Thus, if your application seeks data on particular cities, the database should be partitioned based on the key containing the city. On the other hand, for applications that process information based on date, the date field should be used to partition the data.

Certainly, the partition size should be small enough so that each partition can fit on one host. That allows all the data for a particular key you're looking for (city, date, etc.) to be read more quickly because it's all local to a host. Similarly, you should structure partitions so that all the data you need in each read can be taken from a single file. Because the number of files opened has an impact on performance, you should not make the partitions too small; seek to make them at least several megabytes in size. This might mean grouping multiple dates into files that represent a week or a month, or grouping cities by state.

Hive has a two-tier division of data: you can specify a column on which to make partitions (dividing data logically), and further divide each partition into buckets based on arbitrary hash values.

Dynamic partitioning

When you add data to a data store using Hive, you can specify a particular partition to which the data should be added, such as the partition associated with a single date. This is called *static partitioning*. However, you can also specify the date column as the partition and let Hive determine, for each row, which partition it goes in. This is called *dynamic partitioning*. Hive can also create a new partition if necessary when you use dynamic partitioning. If partitioning along each value (such as one partition per date) would create too many partitions, you can limit the number created.

If you create partitions by processing data with Spark, you can use the flexible `orderBy` method to write partitions that will be efficient

to read. Using that method, start by listing all the columns on which you want to partition data. If you want to further subdivide the data, specify next the columns on which you often filter data. You can then subdivide the data even more by listing high-cardinality columns (that is, columns that contain many different values, such as a unique ID). Spark can estimate the sizes represented by the values in the columns and can partition a skewed distribution so that it's evenly divided among partitions. A smart use of `orderBy` clusters data to make reads faster and avoids having files that are too small.

Because partitioning on the basis of the application is so important, what should you do if different applications need access to the data based on different fields? Creating indexes on those fields will boost performance, if the database supports indexes. (MongoDB, Cassandra, and CouchDB all support indexes, for instance.) Each index adds a small size burden to the database, and a larger time burden that affects each write, so indexes should not be used lavishly. And indexes do not substitute for efficient partitioning. So another option is to duplicate the data, using a data store for each application with the proper partitions.

Choosing the Right Data Processing Engine

In the past, DBAs often added data to their databases manually or by running ad hoc scripts. Employees then used ETL or ELT products to transform the data into a form appropriate for the organization's applications. Organizations exchanged data through file transfers. This can be very efficient for transferring large files using large buffers with very little overhead. The sender might transfer data once a day during a low-volume period such as the middle of the night. Once you have a file on a local system, you can read millions of lines a second, particularly using the `readlines` call offered in many languages to put large chunks of a file into memory before reading individual lines. Another process puts the data into the right schema for the database that stores it.

Best Practice

Don't give up simple file transfers if they work for your data. Even the more modern MapReduce data processing used by Hadoop is file based. Bulk data transfers between different forms of storage can use Apache Sqoop and other tools.

But modern organizations often want data within milliseconds, not days. Furthermore, data that comes over the web or from sensors streams as a constant flow of small updates. Traditional ETL can't be used with such data either.

Often, data is stored exactly as received in its raw format and then processed later. Automation has become indispensable with the growth of new sources, while the introduction of streaming data requires new tools for ingestion and transformation.

This section looks at two sets of modern tools for these tasks: message brokers and streaming processors. Examples of earlier message brokers for low-speed data transfer include RabbitMQ and ZeroMQ, but the speed at which sources send data nowadays has led to new tools with faster processing, notably Kafka, Flume, and Pulsar. Streaming processors include Spark, Storm, and Flink. All these newer tools are the modern equivalents of ETL and ELT. Although they were created by the development teams at various corporations, they are all maintained by the Apache Foundation.

Message brokers and streaming processors accept data from outside sources and output their results to data stores or user applications. Both can perform filtering along the way, but they are used for different purposes. Message brokers do little or no transformation; they simply decide where to send the data. In contrast, streaming processors focus on data transformation or analysis. Message brokers are adept at handling multiple sources and destinations, whereas streaming processors start with a single source and send data at each stage to a single destination. Therefore, data engineers find message brokers good for the initial ingestion of data, and streaming processors good for transforming the raw data after it's ingested to produce processed data that is more useful to the organization.

Successful projects tend to expand their functionality, so there's sometimes no clear line between uses for different tools. Kafka Streams and Pulsar Functions let you run functions over data as you

transfer it, thus mimicking the functionality of Spark and Flink. Flume can also alter data through *interceptors*, which can add or remove headers or use mechanisms as regular expressions to do the more subtle alterations associated with streaming processors. Tools from the older database era also evolve to meet modern needs.

Data Ingestion and Transfer: Message Brokers

Message brokers are the most efficient way to store data, so long as you need little or no transformation. There are several reasons to store the data in its original form and perform any transformations you want later. First, the volume and velocity of incoming data could be so high that you risk dropping data if you take the time to process it. Second, a poorly programmed tool could corrupt your data, and you'll want to be able to return to the raw form to process it correctly. Finally, you might not anticipate some need among your users, and you might need to go back to the raw data and process it differently.

So most likely, you will set up a message broker such as Kafka to ingest data into a raw zone in your data center or the cloud. You can then use a streaming processor, as described in the following section, to run transformations and enrichment requested by users on this data in real time.

Message brokers all operate similarly, although they offer different interfaces and have different architectures.

Each broker accepts streams of data from multiple sources or producers, and sends it to multiple recipients or consumers. The sources are often called *publishers* and the recipients are called *subscribers*, the entire architecture thus being called a *pub/sub* protocol. The data is put on a queue until the subscribers are ready to receive it.

Certain guarantees are provided by message brokers. Messages are always delivered in the order in which the queue receives them. Note that if multiple publishers send data to a broker, the messages may not arrive in the order in which they were generated (in technical terms, delivery is nondeterministic). Usually, therefore, each publisher opens its own set of queues. Each item of data is placed on one queue, and messages on each queue are delivered to subscribers in the order that the messages entered that queue (with various exceptions). The delivery of messages from different queues has a

nondeterministic order, but this is not important because generally each process subscribes to a single queue; multiple processes may be consuming messages in parallel from different queues.

Kafka and Pulsar can replicate a message among multiple nodes in a cluster for fault tolerance. The queues also provide some level of persistence for messages until subscribers dequeue them. If a subscriber is slow or even fails and has to be restarted, the messages remain until they are dequeued or the retention period expires. Pulsar offers tiered storage, moving old data automatically to a slower and cheaper cloud storage service.

Distributed systems must deal with messages that are lost because network packets were dropped or because of other problems they encountered along the way. To handle this uncertainty, some form of acknowledgement is sent by the recipient of each message, and the message broker retransmits messages after waiting a reasonable amount of time for an acknowledgement. Following are the different ways (known as *semantics*) to handle failures and transmissions:

At most once
: The sender just sends each message and forgets about it. If the message doesn't reach the recipient, it is lost. For many situations demanding simplicity and low overhead, lost messages are OK. For instance, the IP and UDP internet protocols use these semantics. But most big data applications would prefer to add some overhead to guarantee delivery.

At least once
: The sender waits for an acknowledgement for a fixed amount of time, then resends the message. The message can therefore be received multiple times by a slow recipient. The sender keeps resending the message a certain number of times, after which the sender assumes that the recipient has failed and gives up.

  These semantics ensure that messages are not lost (unless the recipient fails), but require the recipient to check for duplicates and discard them. Generally, the sender assigns a unique incrementing ID number to each message, and the recipient remembers which IDs it has received so that it can discard duplicates. Kafka calls these IDs *offset numbers* because they represent an offset into the queue, whereas most other systems call them *sequence numbers*.

Exactly once

> The sender retransmits the message as often as necessary, just as in "at least once" semantics, but middleware at the recipient's side checks for duplicates and makes sure that only one copy of each message gets through. Thus, the recipient gets each message exactly once.

Kafka and Pulsar allow the subscriber to choose which of the three semantics to use. Pulsar uses the term "effectively once" in place of "exactly once". Kafka's designers claim that they can provide "exactly once" semantics with only a small degradation of performance. Flume offers only "at least once" semantics.

Message brokers generally also offer encryption through SSL to protect transmissions from snooping or tampering.

Messages are directed to the proper subscribers through a variety of filters. Kafka, probably the most popular message broker for big data, requires the publisher to assign a string or keyword called a *topic*. For instance, an application tracking water quality might label messages with topics such as "bacteria" and "heavy metals" to show what it's reporting in each message. The consumer chooses topics in order to control which messages it receives. Pulsar also uses the term *topic* for the key assigned to direct the messages, but Flume uses a more flexible concept called a *channel* to refer to a queue of messages.

In Kafka, each message can be received by only one consumer. The great benefit of using Kafka is that you can divide messages among multiple consumers, exploiting parallelism to achieve the speed you need. Kafka will break messages from each topic into multiple queues that it calls *partitions* (which should not be confused with the partitions in HDFS or other data stores). Each Kafka partition is associated with a single consumer, so the messages are divided among consumers by being queued on different partitions. (A consumer can, however, read from multiple partitions.)

Flume and Pulsar allow multiple consumers to receive the same message. For instance, suppose you have an application that tracks water quality and records a stream (no pun intended) of messages with readings. One application might be interested only in readings that exceed a certain value, indicating a danger that should be reported to water authorities. Another application may be interested only in messages that record lead levels, regardless of how high or

low they are. And yet another may want all messages, which it stores in a database. The proper messages will be delivered to each application. This is called *multiplexing*.

Sources are called *spouts* in Storm, and data passes from the spouts through one or more *bolts* to be processed. Like Flume, Storm allows multiple stages in a pipeline.

Streaming Data Processing

The processing of streaming data is an important task in today's analytics. Storm came along to enable the era of streaming data processing. Other tools with similar purposes include Flink and Spark, which can handle both streaming and batch processing.

These tools can also be useful in data engineering because you may be responsible for cleaning and prepping data. Streaming processors can be useful for the task of transformation and enrichment (adding provenance data, creating aggregate data, etc.) mentioned in "Data Engineering Today" on page 4. Analytics can help you reduce your data; for instance, your analysis of malfunctions in automobiles might turn up 6,000 possible causes (also called *features* or *dimensions*), and analytics might reveal that you need to save only 12 of them to accurately predict a malfunction.

Streaming tools grab incoming data, whether batch or real time, and transform or run analytics on it. Their programming interfaces exploit method chaining, a common syntax in modern programming languages, to set up pipelines. Spark can ingest data in batches (Datasets or DataFrames) or as a continuous stream through a more recent feature called Spark Streaming. Flink and Storm were designed to work with continuous streams. These tools are widely used by both data scientists and data engineers.

Example Workflow for Streaming Tools

Suppose you have a data set regarding restaurant visits that is updated with new data several times a day, and that you have agreed to perform a sequence of tasks on each data set to give your analysts better data:

1. Add two fields indicating the date and time at which the data set was received (provenance metadata).

2. Convert all text to uppercase for consistency.
3. Remove certain sensitive fields such as credit card information.
4. Check for misspellings by running analytics. If "Towne Grill" appears 600 times in a data set, and "Town Grill" appears three times, the analytics correct the three "Town Grill" instances to "Towne Grill."

The streaming data tools may have built-in functions for some of these tasks. For instance, in Flink you can issue UPPER for uppercase conversion, and one of several date and time functions to add the provenance metadata. Other functions can be written by data scientists or a data engineer who has learned some basic programming in Java or Python.

Development Best Practices

Like programmers, data engineers need a robust process for development, testing, bug fixing, and maintenance. Virtual machines and containers make it easy to set up multiple stages or tiers for your work: development, test, and production. You should also collect metrics that help indicate where you can improve efficiency or the use of your data.

Common Development Tools

To perform your task like a software engineer, you can adapt the popular tools that programmers now use for the tasks of development and deployment:

Version control
: This can manage everything you write in support of your work: code, configuration files, test suites, and documentation. The version-control system ensures that old versions of all these resources can be retrieved quickly in case you find a bug and need to roll back a change. Old versions provide a valuable history, and you can use them to trace when changes were made in case of a problem. They also play a critical role in tying together a team, because everybody has access to the work done by everybody else. You can even use version control to manage contributions from outside your organization and free/open source projects.

Test suites
: Many sophisticated tools allow you to check whether your code carries out what you intended. You can routinely run these tests for *regression testing*: checking for new bugs that are so often introduced by fixes to other bugs or the addition of features. These tests produce error reports in a standard format that lets automatic tools check for and pinpoint problems.

  Many developers need to write applications using databases, as you do. Because setting up these databases can be cumbersome, and queries tend to be slow compared to the tests themselves, testers have created tools called *mocks* that provide synthetic data. Mocks can save you a lot of time during early testing. But they can't simulate everything that can go wrong when writing or querying an actual database. Many problems don't emerge until you are working with full data sets, at the same scale as your production environment. So you will have to perform a part of your testing on a test version of your full data. This also means protecting sensitive data such as credit card numbers in a test environment.

Continuous integration/continuous deployment (CI/CD)
: These tools automatically move your code to the various tiers and run your regression test suites when you check the code into version control.

Bug tracker
: This ensures that problems noted by users or developers are not forgotten. You can also use them to route issues to the proper person who can fix them, and to set priorities.

Integrated development environment (IDE)
: This is a graphical editing environment that recognizes the syntax of the programming language or other files you use. An IDE can save you a lot of time by doing cosmetic formatting, providing autocompletion as you type, checking for syntax errors (which can be annoying if it interrupts your writing, but can prevent a lot of bugs, too), and integrating important tools for testing and development.

Metrics and Evaluation

Data analysis is driving modern organizational decision making, and it can make your work more effective, too.

Collecting metrics is easy—all too easy. Cloud vendors practically urge their metrics on you, and metrics are prolifically available on-premises as well. The first and hardest job is determining what metrics really offer the information that can help your organization run better; you will probably change the metrics you collect as you learn more about the organization's needs.

Cloud vendors live by their metrics, so naturally they offer metrics to clients on every cloud service. AWS integrates all these metrics through CloudWatch, Azure through Azure Monitor, and GCP through a metrics reporting service.

Best Practices

- You can automate the collection of the metrics you choose to monitor and view them on dashboards.
- For on-premises deployments, you can collect metrics using Ganglia and produce graphs or dashboards through a variety of tools. Commercial tools are also available.

Business metrics

It's nearly impossible to make a link between available metrics and big, mission-driven questions such as "Can our employees make decisions that were not open to them before?" or "Did analytics help us improve our annual results?" But organizations can measure some aspects of user behavior, such as:

- How quickly a business user receives the data they request. If this is unsatisfactory, you might need to restructure your data, create new data sets and tables of aggregate data, or add hardware and software resources to speed up transfers.
- How often each data set is being used. A rarely used data set is a waste of money. You have to determine whether it has a purpose within your organization. If it could potentially provide value to

many people who aren't using it, you might need to improve your tags and catalog.
- The speed of queries. This too had implications for data structure, storage, and transmission.

Technical metrics

Business metrics are supported by lower, more technical metrics such as:

- Time required to create a schema and ingest new data
- CPU, memory, and network usage
- Number and sizes of data sets
- Number of queries

Best Practice

Tracking CPU, memory, and network usage can help you specify a budget. But they should be compared to information about the size of data and the number of queries, which can let you know whether you're expending more resources than you should. The combinations of these metrics can point to areas where you need to optimize your data. In a hybrid environment, comparing on-premises metrics to cloud metrics can also help you decide where to place data or look for ways to use your cloud service more efficiently.

Examples of using metrics

Metrics are critical for uncovering failures and poor performance. You should track job failures, which can turn up problems ranging from insufficient memory to programming bugs. The tools used for data engineering jobs, and the goals, are really the same as for any other job.

Usage patterns may also help you with scaling and other planning. Perhaps you ran a resource-intensive background job every night at a time when you expect usage to be low—but one of your users decided to run their own resource-intensive job at the same time, having made the same assumption.

Performance experts distinguish two major types of performance: throughput and latency. Throughput measures the amount of data you can deliver; it's important when a user needs to process a lot of information and can't afford to fall behind. Latency measures the time required for a single message or packet to reach its destination, and is important when a user wants to respond in real-time to a particular stimulus. Often you can improve both throughput and latency by increasing resources. For instance, "Data Ingestion and Transfer: Message Brokers" on page 38 showed how to improve throughput by adding more instances of a streaming processing tool. You can improve latency by increasing physical resources or by moving data closer to the user (such as caching it on the local host).

Sometimes you must trade off between throughput and latency. A good analogy here is a traffic light. During rush hour, the traffic light changes once a minute or so, to let a lot of cars through at once and thus maximize throughput. Late at night when very few cars pass through, the traffic light is configured to change very quickly when a car comes to the red light, because throughput is no longer important and the critical variable is latency.

It's important to experiment with performance changes, because you may be surprised to find out that the impact you expected is not what you actually get. For instance, router manufacturers in the early 2000s reacted to congestion by adding more memory to devices to buffer traffic, but actually slowed down traffic by doing so, a phenomenon identified as *bufferbloat*. More commonly, an administrator tries to fix a problem by increasing some resource that actually was not the cause of the problem, thus wasting the resource while leaving the problem unchanged.

Orchestration

Orchestration means putting in place all the tools and processes you need for smooth operation. To understand what orchestration does, imagine you're an event caterer. You are managing the catered food for a huge wedding party. Ten chefs are busy in the kitchen. As they finish the appetizers, they load the food onto trays and start working on the main course. Meanwhile, you notify five waiters to come and deliver the appetizers. Because several events are being hosted at the same venue, you want to make sure the waiters arrive exactly when the appetizers are ready, but no sooner. (I have never actually

worked for a caterer. This description is not based on reality, but is just a metaphor to explain the concepts in this section.)

In computing, the basic elements of orchestration are *resource management*, *scheduling*, and *fault tolerance* (also known as *high availability*). Let's use the catering metaphor to look at each.

Resource Management

Resource management, as discussed under "Development Best Practices" on page 42, generally involves choosing a server with sufficient CPU, memory, and network bandwidth to run each task. Trial runs can help you determine what resources you need for a particular tool given a particular volume of data. Virtualization lets you specify resources such as CPU and memory in very precise amounts, whether through virtual machines, containers, or the aforementioned online services known as serverless computing. Running on-premises, you might spin up a container with the specified resources. In the cloud, you might start a new instance of some virtual machine.

In the catering metaphor, imagine that some events offer only appetizers, while others have sit-down meals. If the event offers only appetizers, it needs five waiters to circulate among the crowd. For a sit-down meal, it needs eight waiters. So the caterer hires a personnel manager who makes sure that the proper number of waiters are available when needed. If the facility runs out of waiters, some events might have to wait until the waiters are no longer needed by other events. But nobody gets more waiters than they need, so the waiters are always busy.

In computing, the CPU and memory are the waiters. Hadoop offers Yarn to do resource scheduling on top of a Hadoop cluster.

Scheduling

Let's turn now to the scheduling part of orchestration. For data engineering, the equivalent of chefs could be developers checking new tools for reading data into a version-control system, and the waiters could be test suites. The manager in charge of scheduling would be an orchestration tool that creates a workflow connecting the version control system to the test suites. Each check-in to the version control system automatically launches regression tests, which tell you whether you introduced a change that will break the application.

Thus, we are orchestrating a simple CI/CD workflow, a part of the trend that several years ago was labeled DevOps. Such a tool provides advantages by:

- Saving you time and trouble by automating an important step (in this case, running tests).
- Ensuring that tests are run every time they're needed, thus eliminating the potential error of forgetting to run the tests.
- Encouraging a disciplined approach to development and rollout. For instance, knowing that test suites will run reliably, developers will be more likely to write tests. Furthermore, they will standardize error reporting so that the automated tools can determine what happens during the tests.

AirFlow, created at Airbnb and now maintained as a free software project by the Apache Foundation, is a popular orchestration tool in many development shops (not just among data engineers) because it has a rich user interface, lets you define workflows using the popular Python language syntax, and contains many useful hooks. For instance, one hook lets you send email to an administrator to report events such as errors during test runs.

Other scheduling tools exist, but they are less popular at the moment. For example, the Apache Software Foundation offers a workflow tool called Oozie, aimed narrowly at Hadoop and related tools. Furthermore, Oozie is programmed in Java with configuration files in XML, which are more complex than the Python syntax used by AirFlow. AirFlow uses a rather complex syntax, Jinja templates, to format messages for the user, but this is not a part of AirFlow's core features.

Both Oozie and Airflow recognize time and can help you check whether you are meeting your SLAs.

Example: SLA support in AirFlow

Here's how SLA support might work in AirFlow. (We have moved out of testing now and are deploying your tools live.) Suppose you are taking in streaming data and divide the data into chunks called windows as it comes in, starting a new window every five minutes. You want to deliver all the data you received during a five-minute window and start a new task to retrieve the next window. Thus, you can check the time a task starts and ensure that it starts when the

previous task finished. You can also check whether the process storing the data finishes within five minutes so that a new process can start the next window. AirFlow will write an error to a log for each missed SLA, and you can review the logs at regular times or run another process regularly to count how many SLAs were missed.

Some other important features of AirFlow include:

- Parallel tasks, which can be managed through pools of tasks similar to the old typing pools of twentieth-century business offices.
- Alternative tasks. For instance, the SLA logging procedure previously mentioned runs a logging process to emit an error message when an SLA is missed, but doesn't run the logging process if everything is fine. You can also choose among tasks and run the appropriate task based on the outcome of another task. Thanks to the use of Python, loops are also supported.
- Data exchange between tasks. They can send messages through channels called XComs and also share data by setting and retrieving variables. The user interface also allows an administrator to create and manipulate variables that are read by the tasks.
- Triggers and sensors to react to external events. Triggers respond to the results of AirFlow tasks—for instance, a missed SLA. Sensors respond to data from services outside AirFlow, such as the web or HDFS.
- Support for running tasks within Docker and Kubernetes containers.

Schedulers

Let's take scheduling to a higher level. In our catering example, we achieved the important assurance that waiters will take out the food after the chefs cook it. But the facility needs an event planner to make sure everything starts on time. The event planner remembers that the Shah/Banerjee wedding starts at 6 PM, the retirement party for Professor Dinaci at 7 PM, and so on. The event planner is responsible for bringing in the staff to help with each event at the right time. There are also repeating events: for instance, the cleaning staff has to be called in at 9 PM each night. These are not workflow issues and are best handled outside Airflow.

One way to handle scheduling is through operating system schedulers, such as *crontab* on Linux and other Unix-like systems. More sophisticated schedulers are provided as programming libraries, such as Quartz in the Java language, and as part of analytics platforms such as Qubole Scheduler. AirFlow is particularly compatible with a tool called Celery, which sets up task queues. Celery in turn depends on other tools: a message broker such as RabbitMQ, and a system called ZooKeeper for fault recovery.

Fault Tolerance and Checkpoints

It would be tedious to manually check whether every job finishes, especially when the number of jobs runs into the thousands. Many modern systems check automatically for jobs that fail or take too long—it might be impossible to tell the difference because both virtual and physical machines sometimes fail silently—and restart jobs as necessary.

As in database replication, a *checkpoint* or *snapshot* in streaming data preserves a coherent view of the data at a particular point in time, so that if a job fails you can pick up from a recent place instead of repeating the whole job. For instance, in Spark or Spark Streaming you issue the `checkpoint` call to set a checkpoint and can restart a job from the most recent checkpoint.

Checkpoints require more state information in streaming than in database replication, because many aggregate operations require some knowledge of previously processed data. Checkpoints are designed to hide the complexity of saved state and give programmers a simple interface. The administrator configures whether old data is saved or discarded, and whether the state information is stored on the local node or to a distributed filesystem (which imposes more overhead but will preserve the state in case the local node fails).

Conclusion

The evolving data engineering profession is less than two decades old. While researching this report, I discovered a scarcity of online information about the role. Although many popular tools can be used productively in pursuit of data engineering, most current documentation focuses on analytics instead of the storage and data transformation tasks for which a data engineer is responsible. Thus,

not only do you have to learn new skills to be a data engineer, but you have to be creative in applying the advice handed out online.

Data engineering is also a hybrid responsibility. You probably will need to handle both batch and streaming data, both data scientists and business users (who exploit data in very different ways), both SQL access and API access, and both on-premises and cloud storage. These split personalities don't appear to be just transitional, but a long-term feature of your work.

Many tasks of the DBA keep their importance in the age of big data. The data engineer should understand schemas, provisioning and scaling, and other basic concepts from the age of relational databases. You should be comfortable with traditional tools as well as new ones.

The plethora of available tools and storage options also keeps evolving, with last year's champion being this year's outcast. To serve your users, you'll be kept busy looking at new options as they come along. You are not expected to become an expert at using every relevant tool; different members of your team can specialize. And don't take every breathless blog posting seriously when someone claims to have discovered a new solution or to have uncovered a fatal flaw in some old solution. Very little is new under the sun, and if you research the concepts behind a heralded new solution, you will probably find that someone originally thought of it in the 1960s. In particular, the moves from block to object storage and from relational to NoSQL databases, in some respects, rewind computing history.

I'll finish, therefore, by reiterating that understanding your many different users and your organization's needs is critical. Get feedback from users and their managers before you buy tools or transition to a new environment, and at regular intervals thereafter. The right data engineering solution is not necessarily the one getting a lot of buzz, but the one that fits the data and goals of your organization.

Thanks go to our reviewers, particularly Matt Hogan of ManuLife/John Hancock; Nicole Schwartz, product manager for Secure at GitLab; Alex Gorelik, author of The Enterprise Big Data Lake (O'Reilly)*; Veljko Krunic of Krunic Consulting, LLC; Simeon Schwarz, director of data and analytics for OMS National Insurance; and Dr. Hans Donker of Hans Donker Consultancy. Any errors should be attributed to the author alone.*

APPENDIX

Best Practices for Managing Resources

This report has laid out a dizzying platter of tools available for data engineering, but until now has avoided the question of what computer systems to run the tools on. Like any computing operation, you need physical resources in order to do ingestion and transformations. Nearly everyone now schedules resources through either containers or virtual machines (VMs), on-premises or in the cloud, because they allow easy scaling and the efficient exploitation of physical resources.

Containers and Virtual Machines

A container essentially runs a single application in an isolated environment, whereas a VM runs a whole operating system, hosting any applications you want to include. Proponents of VMs claim they are more secure than containers, although attacks against both have been recorded. Containers are more lightweight and can spin up faster.

Platform as a Service (PaaS) is another convenient cloud solution, providing an API on which you can run your programming functions. PaaS makes resource management particularly easy for the programmer, because the vendor handles all of the CPU and memory resources behind the scenes.

All the modern big data tools described in this report have interfaces to popular tools for containers and VMs, both on-premises and among cloud vendors. You need to hook up the tool of your choice to the system of your choice, and specify how much CPU and memory you need for each instance of the job you are running.

For containers, Docker and Kubernetes are the most popular choices. Docker supports load balancing natively, directing all traffic sent to your IP address to what it calls a *swarm*: a collection of containers that you run on separate systems. Kubernetes lets you connect to an external load balancer for the same result. Both systems handle failures by starting up new instances of your job.

VMware is the most popular VM solution for on-premises use, although most sites that use VMs usually reserve them through a third-party vendor. Nimbus is another open source tool that lets you reserve multiple VMs for a job. It must be installed as an agent on each VM, and is controlled by a central broker.

The process for managing resources varies for each process you want to run (Kafka, Spark, etc.) and the container or VM tools you choose. But essentially, processes that you create through Kafka, Spark, or other tools are just like other processes that can be scheduled on the container or VM.

To prevent a single point of failure, the server that controls the various worker processes should be replicated and monitored so that a failure can be repaired by choosing a new server. A cloud service can do this for you. If you run on-premises, you need another open source tool, Zookeeper, to make sure central servers stay alive.

Along with configuring the resources available to containers and VMs, you need to do other configurations such as assigning IP addresses. Naturally, all the tools here offer a wealth of options and a certain amount of orchestration (see "Orchestration" on page 46).

Finally, the network must be configured for high-speed data exchange. This is a huge topic beyond the scope of this report, encompassing such varied disciplines as router hardware choices, routing protocols, and virtual networking.

Operating System Support

Microsoft Windows has been the operating system of choice for most businesses and other organizations for decades, although Unix made inroads during the 1980s. The stabilization of GNU/Linux in the 1990s brought a Unix-style operating system back into the mainstream. All of the big data tools discussed in this report were created by organizations firmly embedded in the world of Linux as well as other free and open source software.

So if you work with the tools in this report, you are likely to end up on Linux systems and will need to get to know them well. Some of the tools release versions for other operating systems, but these should be considered ports of the original Linux version. They probably have not been tested as well as the Linux versions and probably will suffer from bugs and oddities.

About the Author

As an editor at O'Reilly Media, Andy Oram brought to publication O'Reilly's Linux series, the ground-breaking book *Peer-to-Peer*, and the bestseller *Beautiful Code*. Andy has also authored many reports on technical topics such as data lakes, web performance, and open source software. His articles have appeared in *The Economist*, *Communications of the ACM*, *Copyright World*, the *Journal of Information Technology & Politics*, *Vanguardia Dossier*, and *Internet Law and Business*. Conferences where he has presented talks include O'Reilly's Open Source Convention, FISL (Brazil), FOSDEM (Brussels), DebConf, and LibrePlanet. Andy participates in the Association for Computing Machinery's policy organization, USTPC. He also writes for various websites about health IT and about issues in computing and policy.