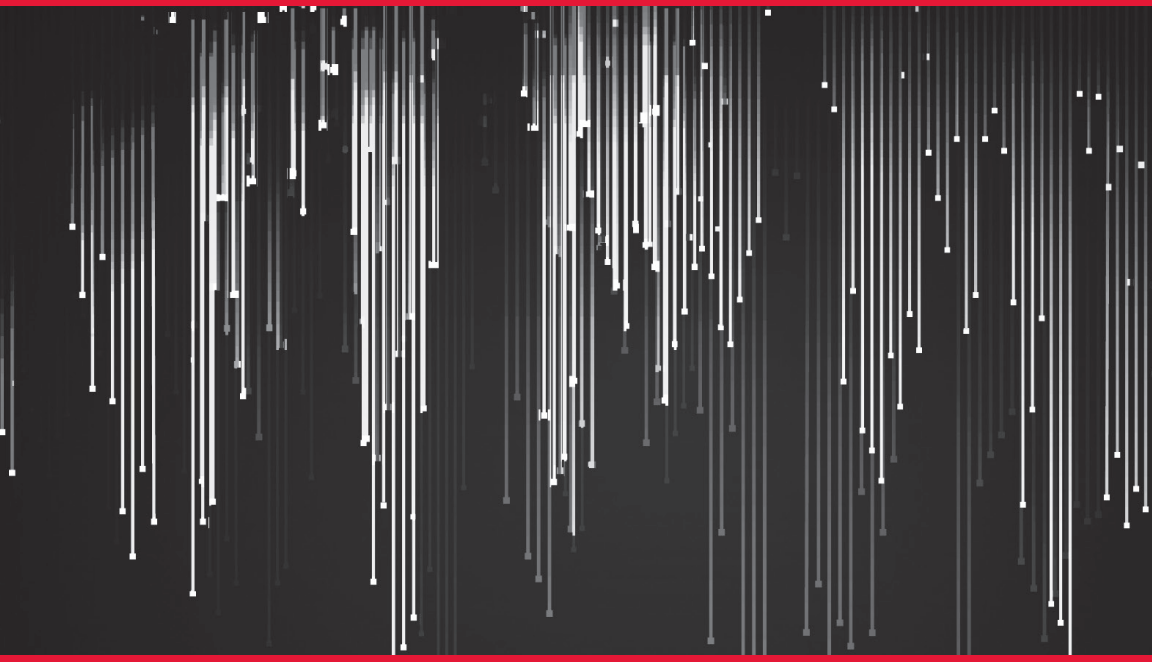
O'REILLY®

Streaming Integration

A Guide to Realizing the Value of Real-Time Data



Steve Wilkes & Alok Pareek

Streaming Integration

A Guide to Realizing the Value of Real-Time Data

Steve Wilkes and Alok Pareek

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

Streaming Integration
by Steve Wilkes and Alok Pareek

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*.

Acquisition Editor: Jessica Haberman
Developmental Editor: Michele Cronin
Production Editor: Katherine Tozer
Copyeditor: Octal Publishing, LLC
Proofreader: Justin Billing
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

February 2020: First Edition

Revision History for the First Edition
2020-02-13: First Release

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

978-1-492-04579-3

[LSI]

Table of Contents

Preface vii

Introduction ix

1. Streaming Integration 1
Real Time 2
Continuous Collection 2
Continuous Movement 2
Any Enterprise Data 3
Extreme Volumes 3
At Scale 4
High Throughput 4
Low Latency 5
Processing 5
Analysis 7
Correlation 9
Continuous Delivery 9
Value 10
Visibility 10
Reliable 11
Verifiable 12
A Holistic Architecture 12

2. Real-Time Continuous Data Collection 15
Databases and Change Data Capture 16
Files and Logs 21
Messaging Systems 22

Cloud and APIs 26
Devices and the IoT 26

3. Streaming Data Pipelines. 31
Moving Data 32
The Power of Pipelines 37
Persistent Streams 38

4. Stream Processing. 41
In-Memory 42
Continuous Queries 43
SQL-Based Processing 44
Multitemporality 46
Transformations 47
Filtering 47
Windows 49
Enrichment 50
Distributed Caches 51
Correlation 52

5. Streaming Analytics. 55
Aggregation 56
Pattern Matching 58
Statistical Analysis 59
Integration with Machine Learning 59
Anomaly Detection and Prediction 62

6. Data Delivery and Visualization. 63
Databases and Data Warehouses 65
Files 68
Storage Technologies 69
Messaging Systems 70
Application Programming Interfaces 72
Cloud Technologies 72
Visualization 74

7. Mission Criticality. 77
Clustering 78
Scalability and Performance 79
Failover and Reliability 82
Recovery 84

Exactly-Once Processing 86
Security 86

8. Conclusion. 91

Preface

It's difficult to imagine our world today without technology. Every aspect of our lives is supported, governed, and enhanced by computing systems and electronic devices. The food we eat, the products we consume, the cars we drive, our health, and the ever-changing world of instant news, information, and entertainment are all created, supplied, optimized, and paid for through digital technologies.

Underpinning this technological revolution is data. Every interaction creates data, and we can use that data for great good – to understand our desires, cure our ills, and generally improve our lives. Those businesses that are not data-driven are feeling competitive pressure to become so. However, it is no longer good enough just to have data. If that data is old, or is not acted on fast enough, it can also spell the downfall of even the best-intentioned organization.

We are in the midst of a vast digital transformation in which even the most conservative enterprises are modernizing through technology – and the data it generates – to optimize their processes, outperform their competitors, and better serve their customers.

But what, exactly, is data, why is it so valuable, and how can your company make the best use of it? An even more important question: what is the best way to modernize the way you use and manage data?

To answer these questions and explain why streaming integration and working with real-time data is such an important part of this modernization, we need to start at the very beginning, and then travel to the future of what's possible.

This book will explain data modernization through *streaming integration* in detail to help organizations understand how it can be applied to solve real-world business problems. We begin with a history of data: what is it? How have we traditionally gathered and worked with it? How do we currently manage the real-time data that is being generated at volumes almost beyond our comprehension? We then introduce the idea of real-time streaming integration: what it is and why it is so critical to businesses today.

In the rest of the book, we examine the steps businesses must take to reap value from streaming integration. We start with building streaming data pipelines and then proceed to stream processing (getting the data into the form that is needed) and streaming analytics. We wrap up the book talking about data delivery and visualization, and, finally, about the mission-critical nature of data.

By the end of this book, you will not only understand the importance of streaming integration to deriving value out of real-time data, you will have a sense of what to look for in a streaming integration platform to realize that value through streaming implementations that address real-world business challenges.

Acknowledgments

We would like to acknowledge several people who helped us make this book a success. First, we would like to express our deep gratitude to Alice LaPlante and the editorial staff at O'Reilly for making the authoring process as easy and painless as possible.

Of course, none of this would be possible without the support and shared vision of our cofounders Ali Kutay and Sami Akbay, and the employees of Striim who work hard to bring the vision of streaming technologies to the enterprise.

We would especially like to thank Katherine Rincon and Irem Radzik for their project management and editorial review of the book. Their objective pairs of eyes helped give us a continuous flow of rich information in a consistent tone.

Finally, we would like to share our appreciation with our families for allowing us to work on yet another project that, at times, took us away from what we enjoy most – being with them.

Introduction

Data has existed since the early Mesopotamians began recording their goods, trades, and money flow, more than seven thousand years ago. Data is quite simply the representation of facts, with a single datum being a single fact. The first data analytics – the process by which data can be translated into information, knowledge, and actions – was most likely the same ancient people determining whether they had a surplus of animals or grains at the end of a season, and using that to decide whether to sell or buy.

The first general-purpose programmable computer designed to work with data was the Electronic Numerical Integrator and Computer, or ENIAC, which powered on in 1945 and was controlled by switches and dials with data fed into it via punch cards. It was used for diverse tasks such as helping to develop the hydrogen bomb, design wind tunnels, and predict weather. It didn't, however, manage or store data. It wasn't until the 1960s that a true data management and processing system, or *database*, would be created.

Although computers had previously been used for automating manual accounting tasks, and complex control systems, the Semi-Automated Business Research Environment (SABRE) airline reservation system of the 1960s was the first true transactional database system. This system ensured single booking of seats and handled more than 80,000 calls per day by 1964.

At that time, data was mostly stored in a hierarchical (document-like) structure. In 1970 Edgar Codd of IBM wrote a paper describing a relational system for storing data, and showed how it could not only handle creating, updating, and deleting data, but be used for querying it. Codd's system consisted of tables that represented

entities, such as organizations and people, and the relationships between the entities. IBM started a research project called System R to implement Codd's vision, and created Structured Query Language, or SQL, as the language for working with data.

Inspired by Codd's vision, Eugene Wong and Michael Stonebraker of the University of California, Berkeley, created the INteractive Graphics REtrieval System (INGRES) as the first commercial SQL-based *relational database management system* (RDBMS), which was distributed as source code to many universities at a nominal cost.

During the 1980s, RDBMSs became increasingly popular. INGRES spawned multiple commercial offerings including Sybase, Microsoft SQL Server, and NonStop SQL, while System R resulted in the IBM SQL/DS (later Db2) and Oracle databases. These databases became the storage and retrieval systems for operational business software applications used for supply chain, inventory management, customer relationships, and others which became packaged together as *Enterprise Resource Planning* (ERP) systems. These *Online Transaction Processing* (OLTP) systems became the backbone of industry.

However, to analyze data and provide businesses with insights in what were known as decision-support systems, a different solution was required. The *data warehouse* was created, starting with Teradata in 1983, to be the home of all enterprise data across multiple systems, with an architecture and data structure that facilitated fast and complex queries. Additional software was created to feed data warehouses using batch processing from operational systems through a process involving the *extraction*, *transformation*, and *loading* (ETL) of data.

Further software emerged that could analyze, visualize, and produce reports on this data, and in 1989 the term *business intelligence* (BI) was used to describe packages from Business Objects, Actuate, Crystal Reports, and MicroStrategy.

This rise of the World Wide Web in the 1990s changed all of this. The interaction of millions, and soon billions, of people across millions of websites generated exponentially more data, and in different forms, than the structured, limited-user, operational business systems. In 2003, the notion of the "Three Vs" – volume, velocity, and variety – of data was coined to express the change in the nature of data the web had introduced. New technology was required to deal

with this, and Hadoop was invented in 2006 as a way to scale data storage and analytics for this new *big data* paradigm.

A Batch of Problems

Databases have thus been the predominant source of enterprise data for decades. The majority of this data came from manual human entry within applications and web pages, with some automation. Data warehouses, fed by batch-oriented ETL systems, provided businesses with analytics. However, in the past 10 years or so, businesses realized that machine data, logs produced by web servers, networking equipment, and other systems could also provide value. This new unstructured data, generated by an ever-increasing variety of sources, needed newer big data systems to handle it as well as different kinds of analytics.

Both of these waves were driven by the notion that storage was cheap and, with big data, almost infinite, whereas CPU and memory were expensive. As a result, the movement and processing of data from sources to analytics was done in batches, predominantly by ETL systems. Outside of specific industries that required real-time actions, such as equipment automation and algorithmic trading, the notion of truly real-time processing was seen as expensive, complicated, and unnecessary for traditional business operations. However, batch processing is crumbling under the strain of competing modern business objectives, shrinking batch windows in a 24/7 world where businesses hunger for up-to-the-second information.

Under Pressure

Business leaders around the world must balance a number of competing pressures to identify the most appropriate technologies, architectures, and processes for their business. Although cost is always an issue, this needs to be measured against the rewards of innovation. Risks of failure versus the status quo must also be considered.

This leads to cycles for technology, with early adopters potentially leapfrogging their more conservative counterparts, who might not then be able to catch up if they wait for full technological maturity. In recent years, the length of these cycles has been dramatically

reduced, and formerly solid business models have been disrupted by insightful competitors – or outright newcomers.

Data management and analytics are not immune to this trend, and the increasing importance of relevant, accurate, and timely data has added to the risk of maintaining the status quo.

Business look at data modernization to solve problems such as:

- How do we move to scalable, cost-efficient infrastructures such as the cloud without disrupting our business processes?
- How do we manage the expected or actual increase in data volume and velocity?
- How do we work in an environment with changing regulatory requirements?
- What will be the impact and use cases for potentially disruptive technologies like artificial intelligence (AI), blockchain, digital labor, and the Internet of Things (IoT), and how do we incorporate them?
- How can we reduce the latency of our analytics to provide business insights faster and drive real-time decision making?

It is clear that the prevalent legacy and predominantly batch method of doing things might not be up to the task of solving these problems, and a new direction is needed to move businesses forward. But the reality is that many existing systems cannot be just ripped out and replaced with shiny new things without severely affecting operations.

Time Value of Data

Much has been written about the "time value of data" – the notion that the worth of data drops quickly after it is created. We can also presume from this notion that if the process of capturing, analyzing, and acting on that information can be accelerated, the value to the business will increase. Although this is often the case and the move to real-time analysis is a growing trend, this high-level view misses many nuances that are essential to planning an overall data strategy.

A single piece of data can provide invaluable insight in the first few seconds of its life, indicating that it should be processed rapidly in a streaming fashion. However, that same data, when stored and aggre-

gated over time alongside millions of other data points, can also provide essential models and enable historical analysis. Even more subtly, in certain cases, the raw streaming data has little value without historical or reference context – real-time data is worthless unless older data is also available.

There are also cases for which the data value effectively drops to zero over a very short period of time. For these *perishable insights*, if you don't act upon them immediately, you have lost the opportunity to do so. The most dramatic examples are detecting faults in, say, power plants or airplanes to avoid catastrophic failure. However, many modern use cases – prevention, real-time offers, real-time resource allocation, and geo-tracking, to name a few – are also dependent on up-to-the-second data.

Historically, the cost to businesses to move to real-time analytics has been prohibitive, so only the truly extreme cases (such as preventing explosions) were handled in this way. However, the recent introduction of streaming integration platforms (as explained in detail throughout this book) has made such processing more accessible.

Data variety and completeness also play a big part in this landscape. To have a truly complete view of your enterprise, you need to be able to analyze data from all sources, at different timescales, in a single place. Data warehouses were the traditional repository of all database information for long-term analytics, and data lakes (powered by Hadoop) have matured to perform a similar function for semi-structured log and device data. If you wanted to analyze the same data in real time, you needed additional systems given that both the warehouse and the lake are typically batch fed with latencies measured in hours or days.

The ideal solution would collect data from all the sources (including the databases), move it into a data lake or scalable cloud data warehouse (for historical analysis and modeling), and also provide the capabilities for real-time analysis of the data as it's moving. This would maximize the time-value of the data from both immediate and historical perspectives.

The Rise of Real-Time Processing

Fortunately, CPU and memory have become much more affordable, and what was unthinkable 10 years ago is now possible. Streaming integration makes real-time in-memory stream processing of all data a reality, and it should be part of any data modernization plans. This does not need to happen overnight, but can be applied on a use-case-by-use-case basis without necessitating ripping and replacing existing systems.

The most important first step enterprises can make today is to utilize streaming integration to move toward a *streaming-first* architecture. In a streaming-first architecture, all data is collected in a real-time, continuous fashion. Of course, companies can't modernize overnight. But the ability to do continuous, real-time data collection enables organizations to integrate with legacy technologies. At the same time, they can reap the benefits of a modern data infrastructure capable of meeting the ever growing business and technology demands within the enterprise.

When data is being streamed, the solutions to the problems mentioned earlier become more manageable. Database-change streams help keep cloud databases synchronized with those on-premises while moving to a hybrid cloud architecture. In-memory edge processing and analytics can scale to huge data volumes and be used to extract the information content from data. This massively reduces its volume prior to storage. Streaming systems with self-service analytics can help companies be agile and nimble, and continuously monitoring systems can ensure regulatory compliance. Of course, new technologies become much easier to integrate if, instead of separate silos and data stores, you have a flexible streaming data distribution mechanism that provides low-latency capabilities for real-time insights.

In summary, data modernization is becoming essential for businesses focused on operational efficiency, customer experience, and gaining a competitive edge. This book will explain in detail data modernization through streaming integration to help you understand how you can apply it to solve real-world business problems.

CHAPTER 1
Streaming Integration

> Streaming integration is the *real-time continuous collection* and *movement* of *any enterprise data*, handling *extreme volumes*, *at scale*, with *high throughput* and *low latency*. *Processing*, *analysis*, *correlation*, and *delivery of data* happen in flight, giving data *value* and *visibility*, in a *reliable* and *verifiable* fashion (Figure 1-1).

Before we go into depth about what is required to achieve streaming integration, it's important to understand each of the concepts emphasized (in italics) in this definition. In this chapter, you learn why each of these ideas is important and how streaming integration is not complete without all of them present.

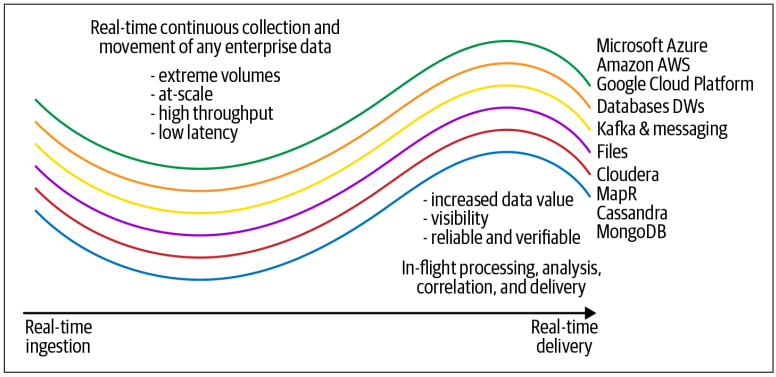


Figure 1-1. Concepts behind streaming integration

Real Time

The overarching principle of streaming integration is that everything happens in real time. There is no delay between data being created, collected, processed, delivered, or viewed such as might be present in traditional Extract, Transform, and Load (ETL) systems or any architecture that uses storage as an intermediary.

This notion means that data must be collected within micro- or milliseconds of being generated. If an organization wants instant insight into its business, any lag will prevent it from understanding what is happening right now.

Continuous Collection

Streaming integration begins with a *streaming-first* approach to data. Unlike ETL, which runs batch processes as scheduled jobs against already stored data, streaming data collection must be continuous (running forever) and in real time. It involves collecting data as soon as possible after it's created and before it ever hits a disk. Although some data sources, like message queues and Internet of Things (IoT) devices, can be inherently streaming and push new data immediately, other sources might need special treatment.

We can think of databases as a historical record of what happened in the past. Accessing data from databases through SQL queries is resource-intensive and uses data already stored on disk. To get real-time information from a database you need a technology called *change data capture* (CDC) that directly intercepts database activity and collects every insert, update, and delete immediately, with very low impact to the database.

Files, whether on disk or in a distributed or cloud filesystem, also cannot be treated as a batch. Collecting real-time file data requires reading at the end of the file and streaming new records as soon as they are written.

Continuous Movement

The result of this continuous collection is a set of data streams. These streams carry the data in real time through processing pipelines and between clustered machines, on-premises and in the cloud. They are used as both the mechanism to continuously

process data as it is created and move it from its point of genesis to a final destination.

For speed and low latency, these streams should mostly operate in memory, without writing to disk, but should be capable of persistence when necessary for reliability and recovery purposes.

Any Enterprise Data

Data is generated and stored in a lot of different ways within an enterprise, and requires a lot of different techniques to access it. We've already mentioned databases, files, message queues, and IoT devices, but that only scratches the surface. Adding to this list are data warehouses; document, object, and graph databases; distributed data grids; network routers; and many software as a service (SaaS) offerings. All of these can be on-premises, in the cloud, or part of a hybrid-cloud architecture.

For each of these categories, there are numerous providers and many formats. Files alone can be written several different ways, including using delimited text, JSON, XML, Avro, Parquet, or a plethora of other formats.

The *integration* component of streaming integration requires that any such system must be capable of continuously collecting real-time data from any of these enterprise sources, irrespective of the type of data source or the format the data is in.

Extreme Volumes

When considering data volumes, figures are often quoted in tera-, peta-, or exabytes, but that is the total amount of stored data. We need to think of data volumes for streaming systems differently. Specifically, we need to consider them in terms of the *rate* at which new data is generated.

The metrics used can be based on the number of new events or the number of bytes created within a certain time period.

For databases, this can be in the order of tens to hundreds of gigabytes per hour stored in transaction logs recording inserts, updates, and deletes – even if the total amount of data stored in the database doesn't change very much. Security, network devices, and system logs from many machines can easily exceed tens to hundreds of

billions of events per day. Many of these logs are discarded after a certain period, so the volume on disk stays fairly constant, but the new data generation rate can be terabytes per day.

There are physical limits to how fast data can be written to disk: on the order of 50 to 100 MBps for magnetic spinning disks, and 200 to 500 MBps for solid-state drives (SSDs). Large-scale enterprise systems often have parallel systems to achieve higher throughput. The largest data generation rates, however, occur when a disk is not involved at all. Reading directly from network ports using protocols such as Transmission Control Protocol (TCP), User Datagram Protocol (UDP), or HyperText Transfer Protocol (HTTP) can achieve higher data volumes up to the speed of the network cards, typically 1 to 10 GBps.

Real-time continuous data collection and the underlying streaming architecture need to be able to handle such data volumes, reading from disk and ports as the data is being generated while imposing low resource usage on the source systems.

At Scale

Scaling streaming integration falls into a number of broad categories, which we dig into deeper later on. Aside from scaling data collection to hundreds of sources, scaling of processing, and handling in-memory context data also needs to be considered.

Streaming integration solutions need to scale up and out. They need to make use of processor threads and memory on a single machine while distributing processing and in-memory storage of data across a cluster. With many distributed nodes within a scaled cluster, it becomes essential that the streaming architecture for moving data between nodes is highly efficient, and can make use of all available network bandwidth.

High Throughput

Dealing with high volumes, at scale, requires that the entire system be tuned to handle enormous throughput of data. It is not sufficient to just be able to keep up with the collection of huge amounts of data as it's generated. The data also needs to be moved, processed, and delivered at the same rate to eliminate any lag with respect to the source data.

This involves being able to individually scale the collection, processing, and delivery aspects of the system as required. For example, collecting and transforming web logs for delivery into a cloud-based data warehouse might require hundreds of collection nodes, tens of processing nodes, and several parallel delivery nodes, each utilizing multiple threads.

In addition, every aspect of the system needs to be tuned to employ best practices ensuring optimal use of CPU and minimal use of input/output (I/O). In-memory technologies are best suited to this task, especially for processing and data movement. However, persistent data streams might need to be employed sparingly for reliability and recovery purposes.

Low Latency

Latency – or how long the results of a pipeline lag behind the data generation – is not directly related to throughput or scale. It is possible to have a throughput of millions of events per second and yet have a high latency (not the microseconds you would expect). This is because data might need to travel through multiple steps in a pipeline, move between different machines, or be transmitted between on-premises systems and the cloud.

If the goal is to minimize latency, it is necessary to limit the processing steps, I/O, and network hops being utilized. Pipelines that require many steps to achieve multiple simple tasks will have more latency than those that use a single step, rolling the simpler tasks into a single, more complex one. Similarly, architectures that use a hub-and-spoke model will have more latency than point-to-point.

A goal of streaming integration is to minimize latency while maximizing throughput and limiting resource consumption. Simple topologies, such as moving real-time data from a database to the cloud, should have latencies in milliseconds. Adding processing to such pipelines should only marginally increase the latency.

Processing

It is rare that source data is in the exact form required for delivery to a heterogenous target, or to be able to be used for analytics. It is common that some data might need to be eliminated, condensed, reformatted, or denormalized. These tasks are achieved through

processing the data in memory, commonly through a data pipeline using a combination of filtering, transformation, aggregation and change detection, and enrichment.

Filtering

Filtering is a very broad capability and uses a variety of techniques. It can range from simple (only allow error and warning messages from a log file to pass through), intermediate (only allow events that match one of a set of regular expressions to pass through), to complex (match data against a machine learning model to derive its relevance and only pass through relevant data). Because filtering acts on individual events – by either including or excluding them – it's easy to see how we can apply this in real time, in-memory, across one or more data streams.

Transformation

Transformations involve applying some function to the data to modify its structure. A simple transformation would be to concatenate `FirstName` and `LastName` fields to create a `FullName`. The permutations are endless, but common tasks involve things like: converting data types, parsing date and time fields, performing obfuscation or encryption of data to protect privacy, performing lookups based on IP address to deduce location or organization data, converting from one data format to another (such as Avro to JSON), or extracting portions of data by matching with regular expressions.

Aggregation and Change Detection

Aggregation is the common term for condensing or grouping data, usually time-series data, to reduce its granularity. This can involve basic statistical analysis, sampling, or other means that retain the information content, but reduce the frequency of the data. A related notion is *change detection* which, as the name suggests, outputs data only when it changes. The most appropriate technique depends on the source data and use case.

Aggregation of data, by definition, occurs over multiple events. As such, the scope of aggregation is usually a window of time or defined by other rules to retain events. Aggregation is therefore more memory intensive than filtering given that thousands or

millions of events need to be kept in memory and aggregation requires some sizing to determine hardware requirements for edge devices.

Enrichment

Enrichment of data can also be essential for database, IoT, and other use cases. In many instances, the raw data might not contain sufficient context to be deemed useful. It could contain IDs, codes, or other data that would provide little value to downstream analysts. By joining real-time data with some context (about devices, parts, customers, etc.), it's turned into valuable information. Real-time enrichment of data streams is akin to denormalization in the database world and typically increases the size of data, not decrease it.

Implementation Options

All of these processing tasks need to be accessible to those who build streaming integration pipelines. And those who build pipelines need to understand how to work with data. Here are some options for how these tasks could be implemented:

- Have individual operators for each simple task, chained to perform processing
- Use a programming language such as Java or Python to code the processing
- Use a declarative language such as SQL to define the processing

It is possible to mix-and-match these techniques within a single pipeline, but if you want to minimize processing steps, maximize throughput, and reduce latency, utilizing a language such as SQL – which compiles to high-performance code transparently – provides a good compromise between ease of use, flexibility, and speed.

Analysis

Streaming integration provides more than just the ability to continually move data between sources and targets with in-stream processing. After streaming data pipelines are in place, it's possible to gain instant value from the streaming data by performing real-time analytics.

This analysis can be in many forms but generally falls into a few broad categories:

- Time-series and statistical analysis
- Event processing and pattern detection
- Real-time scoring of machine learning algorithms

Time-Series and Statistical Analysis

Time-series analysis can be performed naturally on streaming data because streaming data is inherently multitemporal. That is, it can be delineated according to multiple timestamps that can be used to order the data in time. All data will have a timestamp corresponding to when it was collected. In addition, certain collection mechanisms may access an external timestamp, and the data itself can include additional time information.

By keeping a certain amount of data in-memory, or utilizing incremental statistical methods, it is possible to generate real-time statistical measures such as a moving average, standard deviation, or regression. We can use these statistics in conjunction with rules and other context, which themselves can be dynamic, to spot statistically anomalous behavior.

Event Processing and Pattern Detection

Event processing, or what used to be called complex event processing (CEP), enables patterns to be detected in sequences of events. It is a form of time-series analysis, but instead of relying on statistics, it looks for expected and unexpected occurrences. These often rely on data within the events to specify the patterns.

For example, a statistical analysis could spot whether a temperature changed by more than two standard deviations within a certain amount of time. Event processing can utilize that and look for a pattern in which the temperature continues to increase while pressure is increasing and flow rates are dropping, all within a specified amount of time. Event processing is typically used where patterns are known and describable, often derived from the results of previous data analysis.

Real-Time Scoring of Machine Learning Algorithms

Machine learning integration enables pretrained machine learning models to be executed against streaming data to provide real-time analysis of current data. Models could be used for classification, predictions, or anomaly detection. We can use this type of analysis with data that contains many variables, behaves periodically, or for which patterns cannot be specified, only learned.

The great benefit of performing analytics within streaming integration data flows is that the results, and therefore the business insights, are immediate – enabling organizations to be alerted to issues and make decisions in real time.

Correlation

Many use cases collect real-time data from multiple sources. To extract the most value from this data, it might be necessary to join this data together based on the relationship between multiple data streams, such as the way it is correlated through time, data values, location, or more complex associations.

For example, by correlating machine information, such as CPU usage and memory, with information in application logs, such as warnings and response times, it might be possible to spot relationships that we can use for future analytics and predictions.

The most critical aspects of correlation are: first, that it should be able to work across multiple streams of data. Second, it needs to be flexible in the rules that define correlated events and be simple to define and iterate. Ultimately, this means *continuous delivery* must be considered.

Continuous Delivery

After data has been collected, processed, correlated, and analyzed, the results almost always must be delivered somewhere. That "somewhere" could be a filesystem, database, data warehouse, data lake, message queue, or API, either on-premises or in the cloud. The only exception is when the data is being used solely for in-memory analytics.

Writing data should, wherever possible, also be continuous (not batch) and support almost any enterprise or cloud target and data

format. Similar to continuous collection, we should employ parallelization techniques to maximize throughput to ensure the whole end-to-end pipeline does not introduce any lag. An important aspect of delivery is that it should be possible to ensure that all applicable source data is written successfully, once and only once.

Value

The goal of any form of data processing or analytics is to extract business value from the data. The value of data depends on its nature, and it's important to differentiate between *data* and *information*. Although these terms are often used interchangeably, they should not be. Data is a collection of unprocessed facts, whereas information is data processed in such a way as to give it value.

To maximize this value, we need to extract the information content, and with time-sensitive data this needs to happen instantly, upon collection. This can involve filtering out data, performing change detection, enriching it with additional context, or performing analytics to spot anomalies and make predictions. Streaming integration enables this to happen before data is delivered or visualized, ensuring that the value of data is immediately available to the business through visualizations and alerts.

Other patterns for adding value to data include combining both batch and streaming technologies in a single architecture, which has been termed *Lambda processing*. Streaming integration can both feed an append-only data store used for batch analytics and machine learning as well as provide real-time, in-memory analytics for immediate insight. As an extension to this architecture, stream processing can join historical results to add context to streaming data or invoke pretrained machine learning models to span both batch and real-time worlds.

Visibility

As the name suggests, visibility is the way in which we can present data to the user, often in an interactive fashion. This can involve visualizations in the form of charts and tables, combined together in *dashboards*. The dashboards and charts can be searchable, filterable, and provide drilldown to secondary pages. As opposed to more traditional BI software, streaming visualizations frequently show

up-to-the-second information, but can also be rewound to show historical information.

Visibility in the context of streaming integration can mean one of two things:

- Visibility into the data itself and the results of analytics
- Visibility into the data pipelines and integration flows

The former provides insight into business value, whereas the latter gives an operational view of data collection, processing, and delivery, including data volumes, lags, and alerts on anomalous behavior within the data pipeline.

Reliable

It is essential for any system used for mission-critical business operations to be reliable. This means that the system must do what you expect it to do, operate continuously, and recover from failures.

In the scope of streaming integration, it is important to be able to ensure exactly-once processing and delivery of data, independent of the complexity of the flows. All data generated by a source must be collected, processed, and reliably delivered to the target (Figure 1-2). In the case of server, network, system, or other failures, the data flows must recover and continue from where they left off – ensuring that no data is missed and that all processed data is delivered only once.

Additionally, if individual servers in a cluster fail, the system must be capable of resuming data flows on other nodes to ensure continual operations. Ideally, this should all happen transparently to the user without necessitating the intervention of human operatives.

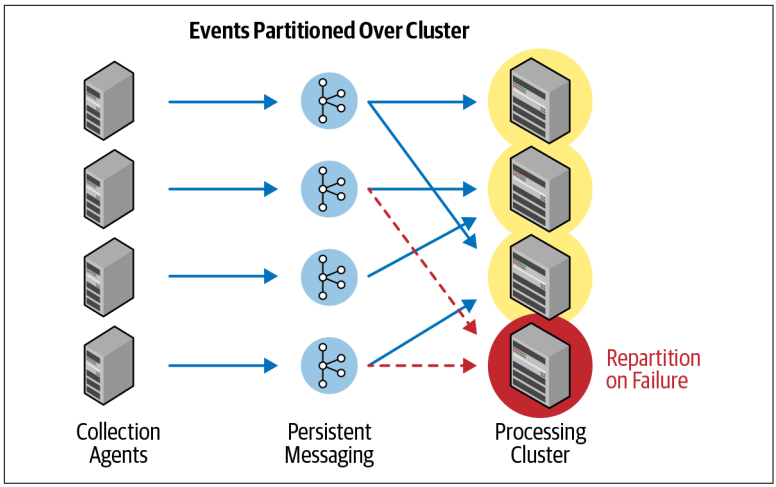


Figure 1-2. Cloud ETL with reliability

Verifiable

Providing reliability guarantees is only half the story. It is also increasingly necessary to be able to prove it and provide insight into the process. Through data flow visibility – including data volumes, number of events, last read and write points, and data lineage – users need to be able to prove that all data that has been read has been both processed and written.

Obviously, this varies by source and target, but the principle is that you need to track data from genesis to destination and verify that it has successfully been written to any targets. This information needs to be accessible to business operations in the form of dashboards and reports, with alerts for any discrepancies.

A Holistic Architecture

In summary, this chapter first defined streaming integration. It then explained its key attributes, which include:

- Providing real-time continuous collection and movement of any enterprise data
- Handling extreme volumes at scale
- Achieving high throughput and low latency

- Enabling in-flight processing, analysis, correlation, and delivery of data
- Maximizing both the value and visibility of data
- Ensuring data is both reliable and verifiable

Streaming Integration should start with a streaming-first approach to collecting data, and then proceed to take advantage of all of these attributes. Any platform that supports streaming integration must provide all of these capabilities to address multiple mission-critical, complex use cases. If any of these attributes are missing, the platform cannot be said to be true streaming integration.

In Chapter 2, we talk about the beginning of the streaming integration pipeline: real-time continuous data collection.

CHAPTER 2
Real-Time Continuous Data Collection

As a starting point for all streaming integration solutions, data needs to be continuously collected in real-time. This is referred to as a *streaming-first approach*, and both streaming integration and streaming analytics solutions cannot function without this initial step. The way in which this is achieved varies depending on the data source, but all share some common requirements:

- Collect data as soon as it is generated by the source
- Capture metadata and schema information from the source to place alongside the data
- Turn the data into a common event structure for use in processing and delivery
- Record source position if applicable for lineage and recovery purposes
- Handle data schema changes
- Scale through multithreading and parallelism
- Handle error and failure scenarios with recovery to ensure that no data is missed

The following sections explain how we can implement these requirements for a variety of different source categories – databases, files and logs, messaging systems, cloud and APIs, and devices and IoT – and will provide examples to clarify each case.

Databases and Change Data Capture

A database represents the current state of some real-world application and is most meaningful in the context of transaction processing. Applications submit queries and updates from a number of network endpoints that are managed as a series of transactions for state observance and transition.

From the late 1970s to the beginning of this century, the term "database" has been commonly used to refer to a relational database in which the underlying entities and relationships between those entities are modeled as tables. Over the past two decades, this term has become an umbrella term for relational database systems as well as with the emerging *NoSQL systems*, which in turn also has become an umbrella term for key-value stores, document stores, graph databases, and others. These databases can be centralized or distributed. They can also be maintained on-premises or stored in the cloud.

However, since databases represent the current state of the data within them, and querying them only returns that state, they are not inherently suited to streaming integration through the query mechanism. Another approach is required to turn the database into a source of streaming data: CDC.

When applications interact with databases, they use inserts, updates, and deletes to manipulate the data. CDC directly intercepts the database activity and collects all the inserts, updates, and deletes as they happen, turning them into streaming events.

CDC Methods

Several CDC methods have been in use for decades, each with its own merits depending on the use case. In high-velocity data environments in which time-sensitive decisions are made, low-latency, reliable, and scalable CDC-powered data flows are imperative.

The business transactions captured in relational databases are critical to understanding the state of business operations. Traditional batch-based approaches to move data once or several times a day introduce latency and reduce the operational value to the organization. CDC provides real-time or near-real-time movement of data by moving and processing data continuously as new database events occur. Moving the data continuously, throughout the day, also uses network bandwidth more efficiently.

There are several CDC methods to identify changes that need to be captured and moved. Figure 2-1 illustrates the common methods.

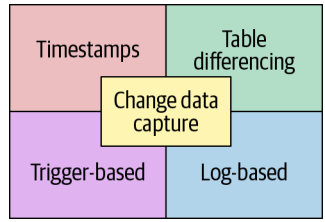


Figure 2-1. Methods of CDC

Let's discuss the advantages and shortcomings of CDC methods:

Timestamps
: By using existing LAST_UPDATED or DATE_MODIFIED columns, or by adding one if not available in the application, you can create your own CDC solution at the application level. This approach retrieves only the rows that have been changed since the data was last extracted. There might be issues with the integrity of the data in this method; for instance, if a row in the table has been deleted, there will be no DATE_MODIFIED column for this row and the deletion will not be captured. This approach also requires CPU resources to scan the tables for the changed data and maintenance resources to ensure that the DATE_MODIFIED column is applied reliably across all source tables.

Table differencing
: By comparing the tables to be replicated in the source and target systems by running a *diff*, this approach loads only the data that is different to enable consistency. Although this works better for managing deleted rows, the CPU resources required to identify the differences is significant and the requirement increases in line with the volume of data. The diff method also introduces latency and cannot be performed in real time.

Triggers
: Another method for building CDC at the application level is defining triggers and creating your own change log in shadow tables. Triggers fire before or after *INSERT*, *UPDATE*, or *DELETE* commands (that indicate a change) and are used to create a change log. Operating at the SQL level, some users pre-

fer this approach. However, triggers are required for each table in the source database, and they have greater overhead associated with running triggers on operational tables while the changes are being made. In addition to having a significant impact on the performance of the application, maintaining the triggers as the application change leads to management burden. Many application users do not want to risk the application behavior by introducing triggers to operational tables.

Log-based CDC
: Databases contain transaction (sometimes called redo) logs that store all database events allowing for the database to be recovered in the event of a crash. With log-based CDC, new database transactions – including inserts, updates, and deletes – are read from source databases' transaction or redo logs. The changes are captured without making application-level changes and without having to scan operational tables, both of which add additional workload and reduce the source systems' performance.

Log-Based CDC Best Suited for Streaming Integration

CDC and, in particular, log-based CDC (Figure 2-2), has become popular in the past two decades as organizations have discovered that sharing real-time transactional data from Online Transaction Processing (OLTP) databases enables a wide variety of use-cases. The fast adoption of cloud solutions requires building real-time data pipelines from in-house databases in order to ensure that the cloud systems are continually up to date. Turning enterprise databases into a streaming source, without the constraints of batch windows, lays the foundation for today's modern data architectures.

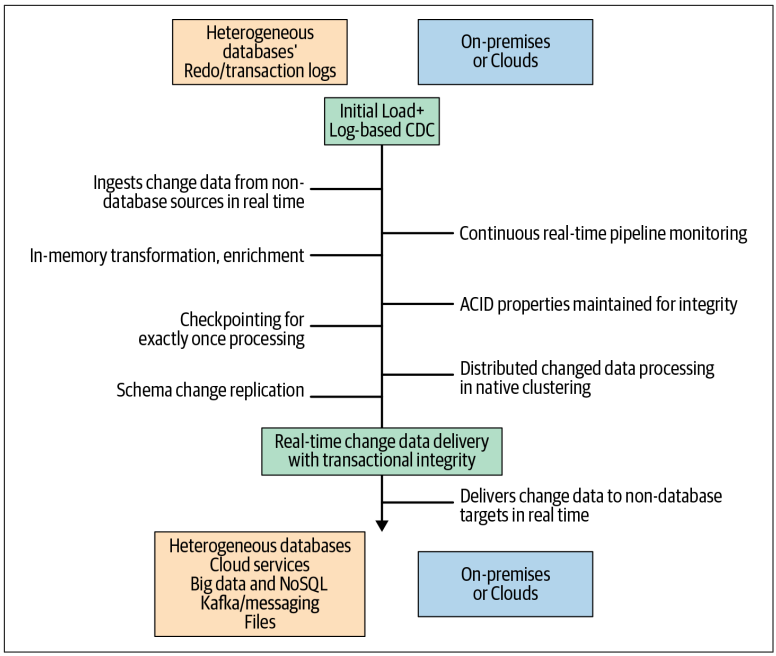


Figure 2-2. Log-based CDC

Streaming integration should utilize log-based CDC for multiple reasons. It minimizes the overhead on the source systems, reducing the chances of performance degradation. In addition, it is nonintrusive. It does not require changes to the application, such as adding triggers to tables would do. It is a lightweight but also a highly performant way to ingest change data. Although Data Manipulation Language (DML) operations (inserts, updates, deletes) are read from the database logs, these systems continue to run with high performance for their end users.

The ingestion of change data through CDC is only the first, but most important, step. In addition, the streaming integration platform needs to incorporate the following:

- Log-based CDC from multiple databases for nonintrusive, low-impact real-time data ingestion to minimize CPU overhead on sources and not require application changes.
- Ingestion from multiple, concurrent data sources to combine database transactions with semi-structured and unstructured data.

- End-to-end change data integration, including:

  Zero data loss guarantee
  : Data loss cannot be tolerated by a downstream application due to the nature of data tracked in database systems. This means that if the external database system or the CDC process fails, event checkpointing must guarantee oldest active events of interest are carefully tracked by the CDC process.

  Event delivery guarantees
  : Exactly-once processing (E1P) and/or at-least-once processing guarantees must be preserved. This requires an understanding of consuming systems and the atomicity semantics they support.

  Ordering guarantees
  : Events are propagated in commit order or in generation order. So, data that's generated in transactional order must be able to retain that order and at source side transactional boundaries as required.

  Transaction integrity
  : When ingesting change data from database logs, the committed transactions should have their transactional context maintained. Throughout the whole data movement, processing, and delivery steps, this transactional context should be preserved so that users can create reliable replica databases.

  In-flight change data processing
  : Users should be able to filter, aggregate, mask, transform, and enrich change data while it is in motion, without losing transactional context.

  Schema change replication
  : When a source database schema is modified and a Data Definition Language (DDL) statement is created, the streaming integration platform should be able to apply the schema change to the target system without pausing.

- Turning change data to time-sensitive insights. In addition to building real-time integration solutions for change data, it should be possible to perform streaming analytics on the change data to gain immediate insights.

Log-based CDC is the modern way to turn databases into streaming data sources. However, ingesting the change data is only the first of many concerns that streaming integration solutions should address.

Files and Logs

Many applications such as web servers, application servers, IoT edge servers, or enterprise applications continually generate data records that are written to files or logs. These files can be on a local disk subsystem, a distributed filesystem, or in a cloud store.

This data contains valuable information needed for operational analytics. In batch processing Extract, Transform, and Load (ETL) systems, these files are written to and closed before being read by ETL. However, for real-time systems it is essential to be able to perform real-time data collection on files that are currently being written to (open files).

Data Collection from Filesystems

Collecting real-time file data requires a set of algorithms to detect changes to the files/directories/nodes:

- Understanding the contents of the underlying file formats to be able to parse the file records
- Maintaining position offsets to reflect the current EOF (end of file) markers for subsequent collection
- Identifying torn/partial records
- Recovery handling to address various failure scenarios

Traditional ETL has successfully managed to extract data from files after a file is complete. But for real-time processing, new records need to be collected as soon as they are written to keep the propagation latency at a lower granularity than the file size.

There are several common patterns that occur in real-time stream processing during ongoing file generation that need to be supported and pose significant technical challenges. Some examples include the following:

- Support for multiple filesystems including Linux (ext*), Windows (NTFS), Hadoop (HDFS), network based (NFS), Cloud Storage systems (AWS S3, Azure ADLS, Google GCS, etc.).
- Support for multiple file formats such as JSON, DSV, XML, Avro, Thrift, Protocol Buffers, and Binary.
- Support for reading from multiple directories and subdirectories from which files need to be read. It's not always possible to have one central repository where all files can be generated.
- Support for data parsing using static and dynamic record delimiters.
- Support for data collection using wildcarding at the levels of files and directories.
- Support for data collection when files are in sequence and rollover to the base sequence.
- Managing the number of open file descriptors.
- Event guarantees with respect to data loss, at-least-once, or at-most-once processing.
- Handling schema changes.

Messaging Systems

Of all the types of sources that can provide data for streaming integration, messaging systems are the most natural fit. They are inherently real time, and push data to consumers. In fact, messaging systems are often a required component of a streaming integration solution, being necessary for the continuous movement of data.

Messaging systems usually consist of *producers* that deliver messages to *brokers* to be read by *consumers*. To continuously collect data from a messaging system, the streaming integration solution needs to be able to connect to a broker as a consumer.

With the rapid increase in the adoption of cloud technologies in the past few years, messaging systems have also been introduced by cloud providers. Microsoft Azure Event Hubs, Amazon Kinesis, and Google Pub/Sub all provide cloud-based messaging platforms that are designed to elastically scale and support streaming and message-driven applications in the cloud.

Because heterogeneous integration and the collection of data from any enterprise (or cloud system) is an essential part of streaming integration, you need to consider all of these different types of messaging systems. Scalability of continuous collection is key given that most of these systems can handle tens of thousands to millions of messages per second.

Data Collection from Messaging Systems

There are two major considerations when working with messaging systems. First, the system needs to connect to the messaging provider and subscribe to receive messages utilizing some API. There are often security, compression, encryption, and architectural pieces to this that need to be resolved within a messaging adaptor.

Second, data needs to be extracted from the message. In addition to a data payload that can be in text, binary, key-value, or other forms, there are additional system and header properties that can contain useful information.

Different messaging systems require different APIs. Aside from Kafka, which has its own API, a good majority of messaging systems support the JMS API or AMQP protocol.

Collecting data from Java Message Service systems

When connecting to Java Message Service (JMS) systems, you need to first create an *initial context* that contains information about connecting to the provider, such as broker URL and security credentials. This information varies by provider. From this, you can obtain a *connection factory* either directly, or by looking up the service through Java Naming and Directory Interface (JNDI). The factory then allows you to create a connection to the provider and create a session through which you will send and receive messages.

For data collection, the interest is in receiving messages and these can be either from queues or topics. Queues are typically point-to-point and only one consumer will receive a message sent to a queue. Topics provide a publish/subscribe topology in which every subscriber will receive a copy of a published message. Queues and topics each have their own separate issues in the areas of scalability and reliability.

Because queues allow only a single consumer to receive a copy of a message, it is impossible to use an existing queue as a data source without breaking any existing data flow. Instead additional queues (or topics) need to be added with existing messages also routed to these new destinations.

Reading from queues has delivery guarantees that will ensure all messages are seen, but this might require *persistent* options to handle failure scenarios. Topics are more suited for data collection because they can have multiple subscribers. However, it is important that such subscribers are *durable*. This means that messages are kept until every subscriber has received them; otherwise they will just be discarded.

The biggest problem with collecting JMS data is recovery. Although JMS supports transactions, it does not permit repositioning, or rewinding within a queue or topic. In complex stateful processing pipelines utilizing windows or event buffers, recovery often requires replaying old events, which is not possible using the JMS API.

Collecting data from Apache Kafka

Apache Kafka is a high-throughput distributed messaging system. It utilizes a publish/subscribe mechanism and is inherently persistent, writing all messages to a distributed commit log. Clients connect to *brokers* as either *producers* or *consumers*, with producers sending messages to *topics* and consumers receiving them as subscribers to that topic. When a producer sends a message, it is stored in an append-only log on disk. The broker can be clustered over a large number of machines, with the data partitioned and replicated over the cluster.

When producers send a message to a broker, a partition key is used to determine which partition, and therefore which machines in a cluster, need to write the data to the log, with each partition written to a separate physical file. The broker can write the data to one or more machines for reliability and failover purposes. The logs are retained for a period of time, and consumers manage their own read location in the log. This enables consumers to come and go, and run at their own speeds without affecting other consumers.

Consumers belong to a consumer group, with each consumer in a group being assigned to one or more partitions. Each consumer group subscribed to a topic will receive all the messages sent to that

topic, but the individual consumers in that group will receive only those messages belonging to its partitions. There cannot be more consumers than partitions, so deciding on a partitioning scheme for a topic is an essential early consideration. Importantly, because each consumer needs to keep track of the log position it is reading from, it is possible for consumers to position backwards and replay old messages, as long as they are retained on disk.

When collecting data from Kafka, it is important to consider both scalability and reliability scenarios.

Scalability of reading from Kafka is directly related to the number of partitions specified for a topic. To read in-parallel from a topic using multiple consumers, it is necessary to have at least as many partitions as consumers. It is possible to add additional partitions to a topic later, but this affects only new data, and it is impossible to reduce the number of partitions. Adding new consumers to a group dynamically (either as additional threads or in separate processes or machines) up to the partition limit enables more data to be read in parallel.

The major difference between Kafka and the other messaging systems is that Kafka requires consumers to track their read positions. This helps with reliability considerations because, in the case of failure, consumers can not only pick up where they left off, but can also rewind and replay old messages. By tracking the read position of consumers, and understanding how far those messages have progressed through a processing pipeline, it is possible to determine how far back consumers need to rewind to rebuild state before processing can resume.

Handling different data formats

The messaging systems described previously have different approaches to understanding the data being transmitted. JMS supports several types of messages including raw bytes, a serialized Java Object, text, and name/value pairs. Both AMQP and Kafka inherently send data as raw bytes, but AMQP can also specify the content-type in a way consistent with HTTP, whereas Kafka can utilize a separate *schema registry* to define the data structure of messages on a topic.

In most practical situations, however, the data is text serialized as bytes and formatted as delimited data, log file entries, JSON, or

XML. From a collection perspective, it is important, then, to enable flexible parsing of text (similar to files) as part of working with messaging systems.

Cloud and APIs

Increasingly, enterprise applications are being deployed in the cloud in a SaaS multitenant pattern. Apps like Salesforce's Sales Cloud, Workday Payroll, and HCM offer a subscription-based model to streamline business processes and enable rapid business transformations, and promise to incorporate newer intelligence in their applications using AI and machine learning. Many enterprises thus are gradually adopting a hybrid cloud deployment model in which newer applications are moving to the cloud.

Rarely will all of a corporation's business applications run on a single public cloud. Often there's a compute grid that spans multiple clouds and on-premises systems, across operations and analytics environments. To gain real-time visibility, data from these cloud SaaS applications also needs to be made available in a streaming manner. In fact, if on-premises systems are set up to receive streaming changes from on-premises applications, a SaaS checklist must include requirements to get data transferred in real time from the SaaS environment.

Some of the techniques we have discussed in previous sections might not pertain to SaaS environments because of inaccessibility of underlying platforms/databases due to security considerations (for example, the opening of certain network ports), service-level agreement (SLA) requirements (ad hoc CDC initial loads), or multitenancy manageability concerns (special trigger handling for CDC). Generally, data for business objects is made available in a batch via a bulk API or in real time via streaming APIs.

Devices and the IoT

The IoT has garnered a lot of attention as a big driver of digital transformation within many industries. Simply put, IoT is the worldwide collection of devices, sensors, and actuators that can collect, transfer, and receive data over a network without requiring human interaction. The "things" in IoT can refer to the devices

themselves, or the objects they are monitoring, including people, animals, vehicles, appliances, and machinery.

Despite the reference to the "internet" in the name, IoT does not need to transfer data over the web. Internet here is a reference to Internet Protocol (IP) which enables data packets to be delivered from a source to destination based only on IP addresses. IoT works across any IP-based network, facilitating internal as well as external use cases. The notion that IoT requires devices to deliver data to the cloud is restrictive given that many commercial, industrial, and healthcare use cases keep the data private.

Data Collection from IoT Devices

The term *IoT device* is very broad and encompasses a wide range of hardware. A single temperature sensor sending data over WiFi can be considered an IoT device. However, a device that includes a temperature sensor, alongside other measurements and logic, such as a smart thermostat, weather station, or fire alarm, could also be an IoT device. Devices can be further combined to produce larger-scale "devices," such as connected cars, smart refrigerators, or home security and control systems.

The size and electrical power available for the device will dictate to some degree how much computing power the device has and the types of protocol it can support. Smaller devices tend to have very little memory or CPU capabilities and require *lightweight* protocols for transmitting data. Larger devices can do more processing, utilize more complex code, and support heavier-weight, more resilient protocols. Figure 2-3 shows streaming integration for IoT.

The simplest protocols used by IoT are TCP and UDP at the transport layer of the TCP/IP network model, sending data directly as network packets to a destination. At the application layer, existing protocols can be used, and new protocols have emerged. HTTP and HTTPS (secure HTTP) are common, often implemented as JSON being sent over Representational State Transfer (REST) calls.

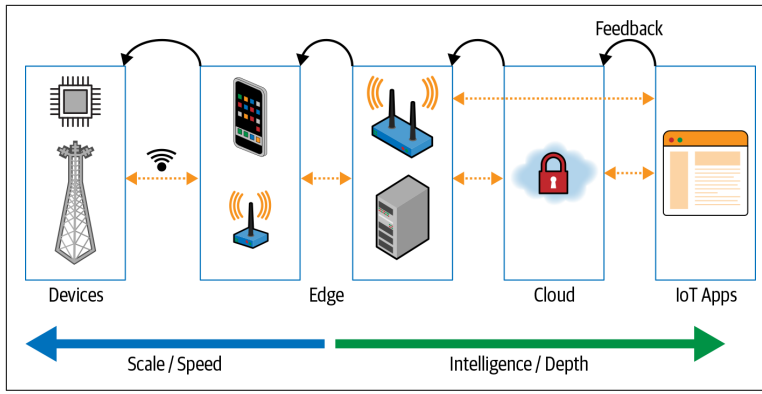


Figure 2-3. Streaming integration for IoT

Message Queuing Telemetry Transport (MQTT) and WebSockets are common publish/subscribe protocols allowing two-way communication with devices. OPC-UA (OPC Unified Architecture from the OPC Foundation) is a next-generation standard that defines a client/server protocol mainly for industrial applications, utilizing UDP or MQTT under the hood for data transport.

Aside from the transfer protocol, the other consideration is the data format. There is no real standard for IoT devices, so integration needs to be considered on a case-by-case basis. JSON is common, but data can also be binary, delimited, XML, or appear in a proprietary textual form.

IoT Scalability Considerations

Any discussion of IoT data almost always includes the concept of *edge processing*. Edge processing is when computation is placed as close as possible to the physical edge device – usually in it – making the IoT device as "smart" as possible. IoT is fairly unique in the data world in how much data it generates, in that there can be hundreds, thousands, or even millions of individual devices all generating small amounts of data. Even if a single sensor or device generates data only 10 times per second, when this is multiplied by the number of devices, it can quickly become overwhelming.

A lot of this data is repetitive, redundant, or just not that interesting. It is the information content present in that data that is really required. (See the previous discussion on Value for the difference between data and information.)

A simple example is a temperature sensor. If a single sensor reads the temperature 10 times per second, it will generate 36,000 *data* points per hour. If the temperature stays at 70 degrees throughout that time, the *information* content is one item: "70 degrees for an hour."

To reduce the amount of data generated by IoT, data from multiple individual sensors can be collected through a single edge device. Here, the data can be filtered, aggregated, and transformed to extract the information content. The important thing here is to not only do statistical analyses and send summary information, but also to be able to react instantly to change. Most edge processing is a combination of statistical summarization plus immediate change detection and sensor health signals.

Using the temperature sensor example, a combination of statistical summarization (min, max, average, standard deviation, etc.) within a specific period, combined with pings every minute to indicate the sensor is alive, and immediate messages if the temperature changes dramatically (plus/minus two standard deviations above mean) will drastically reduce the amount of data that needs to be collected.

The types of processing that can be done at the edge can be quite complex, even incorporating machine learning and advanced analytics for rapid response to important events. This is discussed in more detail in Chapter 6.

CHAPTER 3
Streaming Data Pipelines

After data is collected from real-time sources, it is added to a data stream. A stream contains a sequence of events, made available over time, with each event containing data from the source, plus metadata identifying source attributes. Streams can be untyped, but more common, the data content of streams can be described through internal (as part of metadata) or external data-type definitions. Streams are unbounded, continually changing, potentially infinite sets of data, which are very different from the traditional bounded, static, and limited batches of data, as shown in Figure 3-1. In this chapter, we discuss streaming data pipelines.

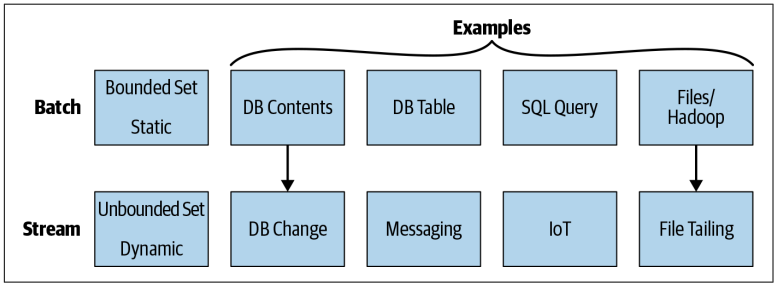


Figure 3-1. Difference between streams and batches

Here are the major purposes of data streams:

- Facilitate asynchronous processing
- Enable parallel processing of data
- Support time-series analysis

- Move data between components in a *data pipeline*
- Move data between nodes in a clustered processing platform
- Move data across network boundaries, including datacenter to datacenter, and datacenter to cloud
- Do this is a reliable and guaranteed fashion that handles failure and enables recovery

Streams facilitate *asynchronous* handling of data. Data flow, stream processing, and data delivery do not need to be tightly coupled to the ingestion of data: these can work somewhat independently. However, if the data consumption rate does not match the ingestion rate, it can lead to a backlog that needs to be dealt with either through *back-pressure*, or *persistence*, which we cover in detail later in this chapter.

Streams also enable parallel processing of data. When a logical data stream is present across multiple nodes in a clustered processing platform, the node on which a particular event will be processed can be determined through a stream partitioning mechanism. This mechanism utilizes a *key*, or other feature of the data to consistently map events to nodes in a deterministic and repeatable manner.

The data delivered to streams is often *multitemporal* in nature. This means that the data might have multiple timestamps that can be used for time-series analysis. Timestamps might be present in the original data, or metadata, or can be injected into the stream event at the time of collection or processing. These timestamps enable event sequencing, time-based aggregations, and other key features of stream processing.

Let's begin our examination of streams through (perhaps) their most important function: moving data between threads, processes, servers, and datacenters in a scalable way, with very low latency.

Moving Data

The most important thing to understand about a stream is that it is a *logical entity*. This means that a single named stream can comprise multiple physical components running in different locations; it has a logical definition, and a physical location. The stream is an abstraction over multiple implementations that enable it to move data efficiently in many different network topologies.

To understand the various possibilities, let's use a simple example of a source reader collecting data in real time and writing it to a stream. A target writer reads from this stream and delivers the data in real time to a destination.

Figure 3-2 illustrates the components involved in this simple data flow, and Table 3-1 provides a description of each.

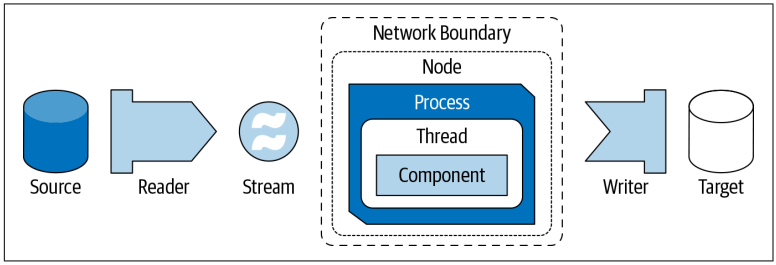


Figure 3-2. Elements of a simple streaming data flow

Table 3-1. Components of a streaming data flow

Source	The origin of real-time data; for example, the database, files, messaging, and so on
Reader	Collects real-time data from the source and writes to a stream
Stream	The continuous movement of data elements from one component, thread, or node, to the next one
Network	Delineates different network locations; for example, on-premises and cloud
Node	A machine on which processes run
Process	An operating system process running on a node with potentially many threads
Thread	An independent and concurrent programming flow within a process
Component	An item running within a thread that can interact with streams
Writer	Receives real-time data from a stream and writes to a target
Target	The destination for real-time data, for example, database, Hadoop, and so on

In all cases, the reader will write to a named stream, and the writer will receive data from the same named stream. The simplest way in which this flow could work is that everything runs within a single thread, in a single process, on a single node, as depicted in Figure 3-3.

The stream implementation in this case can be a simple method (or function) call given that the reader is directly delivering data to the writer. The data transfer through the stream is synchronous, and no serialization of data is required because the reader and writer operate in the same memory space. However, the direct coupling of components means that the writer must consume events from the reader

as soon as they are available but cannot write concurrently with reading. Any slowness on the writing side will slow down reading, potentially leading to lag.

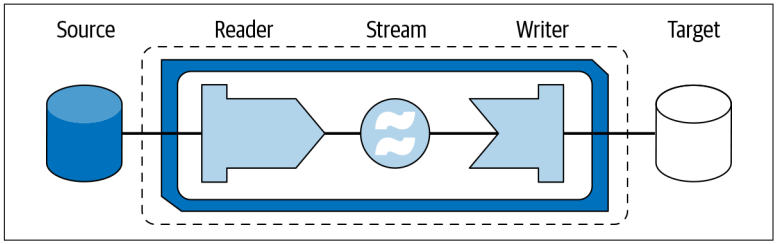


Figure 3-3. A single-thread named stream

To enable concurrency, a multithreaded model is required, with reader and writer operating independently and concurrently.

The stream in this case needs to span threads, and is most commonly implemented as a queue. This queue can be in-memory only or spill to disk to as necessary to handle sizing requirements. The reader and writer can now run asynchronously and at different speeds with the stream acting as a buffer, handling occasional writer slowness to the limit of the queue size. As with the single-threaded mode, no data serialization is required.

In multithreaded applications, the operating system can cause bottlenecks between threads. Even in a multicore or multi-CPU system, there is no guarantee that separate threads will run on different cores. If reader and writer threads are running on the same core, performance will be no better, or even worse than a single-threaded implementation.

Multiprocess models can help with this, using processor affinity to assign CPU cores to particular processes.

In this case, the reader and writer are running in different operating system processes, so the stream needs to span the memory space of both. This can be done in a number of ways, utilizing shared memory, using Transmission Control Protocol (TCP) or other socket connections, or implementing the stream utilizing a third-party messaging system. To move data between processes, it will need to be serialized as bytes, which will generate additional overhead.

The natural extension of this topology is to run the reader and writer threads on separate nodes, with the stream spanning both locations, as demonstrated in Figure 3-4.

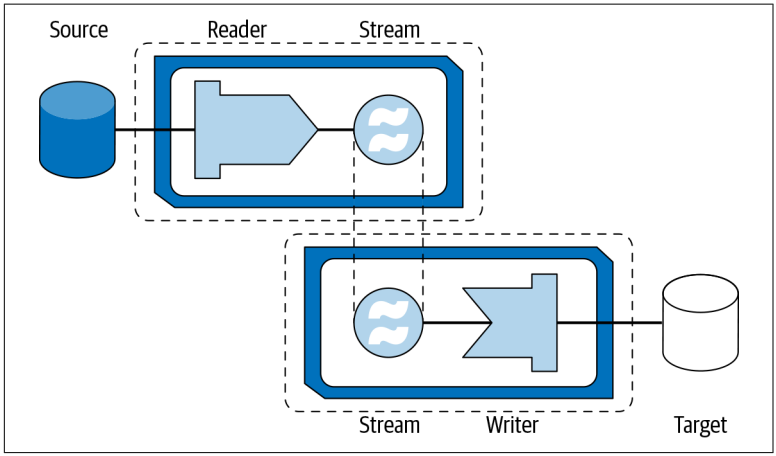


Figure 3-4. Running reader and writer threads on separate nodes

This ensures full processor utilization but eliminates the possibility of using shared memory for the stream implementation. Instead, the stream must use TCP communication, or utilize a third-party messaging system. As with the previous example, data must be serialized as bytes to be sent over the wire between nodes. Latency over TCP between nodes is higher than between processes, which can increase overall data flow latency. This topology is also useful for the case in which sources or targets are accessible only from a particular physical machine. These nodes could be running in the same network domain, or spanning networks, in an on-premises-to-cloud topology, for example.

Spanning networks can introduce additional requirements to the stream implementation. For example, the on-premises network might not be reachable from the cloud. There might be firewall or network routing implications. It is common for the on-premises portion to connect into the cloud, enabling data delivery, but not the other way around.

Streams also enable parallel processing of data through partitioning. For cases in which a single reader or writer cannot handle the real-time data generation rate, it might be necessary to use multiple instances running in parallel. For example, if we have CDC data

being generated at a rate of 100,000 operations per second, but a single writer can manage only 50,000 operations per second, splitting the load across two writers might solve the problem.

We must then carefully think through how the way the data is partitioned. After all, arbitrary partitioning could lead to timing issues and data inconsistencies, as two writers running asynchronously can potentially lead to out-of-order events.

Within a single node and process, we can achieve parallelism by running multiple writer threads from the same stream, as shown in Figure 3-5.

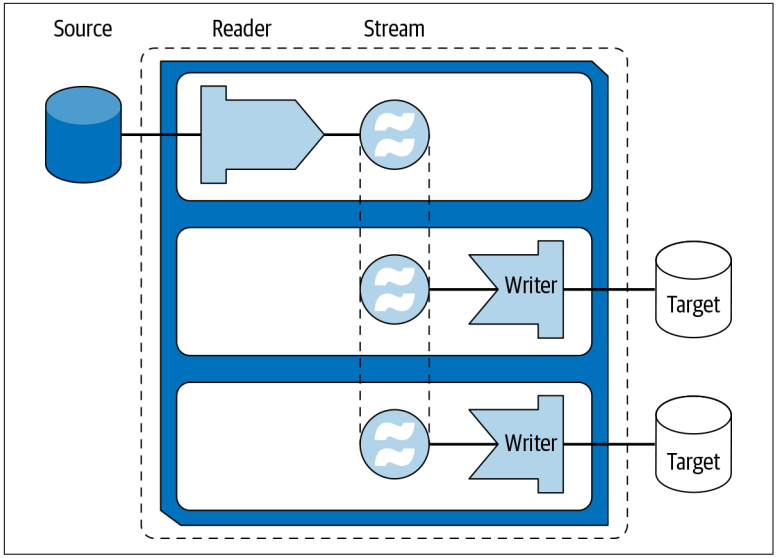


Figure 3-5. Achieving parallelism by running multiple writer threads from the same stream

Each thread will receive a portion of the data based on a partitioning scheme, and deliver data to the target simultaneously. The maximum recommended number of writer threads depends on a number of criteria, but should generally not be greater than the number of CPU cores available (minus one core for reading), assuming threads are appropriately allocated (which usually they are not). The stream should take care of delivering partitioned data appropriately to each thread in parallel.

For greater levels of parallelism, it might be necessary to run multiple writer instances across multiple nodes.

Again, the stream needs to take care of partitioning data: in this case, sending it to different nodes based on partitioning rather than separate threads. It should also be possible to combine the two parallelism mechanisms together to have multiple threads running on multiple nodes to take best advantage of available CPU cores. The level of parallelism possible will depend greatly on the nature of the data, and the requirement for continual consistency.

For example, it might be possible to partition CDC data by table, if the operations acting on those tables are independent. However, if changes are made to related tables (e.g., the submission of an order that makes modifications to multiple tables), the resulting events might need to be handled in order. This might require partitioning by customer or location that all related events are processed in the same partition.

These examples have dealt with the simple case of reading data from a source and writing to a target. It should be clear that there are many possible implementation options within even this basic use case to deal with throughput, scale, and latency. However, many real-world use cases require some degree of stream processing, which requires multiple streams and the notion of a pipeline.

The Power of Pipelines

A streaming data pipeline is a data flow in which events transition through one or more processing steps that progress from being collected by a "reader," and delivered by a "writer." We discuss these processing steps in more detail later in the book, but for now it is sufficient to understand these steps at a high level. In general, they read from a stream and can filter, transform, aggregate, enrich, and correlate data (often through a SQL-like language) before delivery to a secondary stream.

Figure 3-6 presents a basic pipeline performing some processing of data (for example, filtering) in a single step between reader and writer.

We can expand this to multiple steps, each outputting to an intermediate stream, as depicted in Figure 3-7.

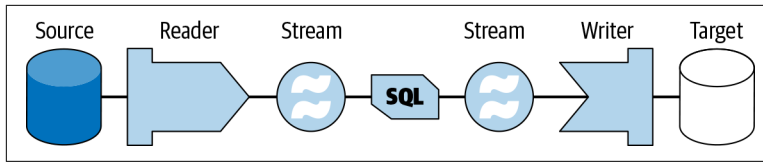


Figure 3-6. Basic pipeline performing filtering in a single step

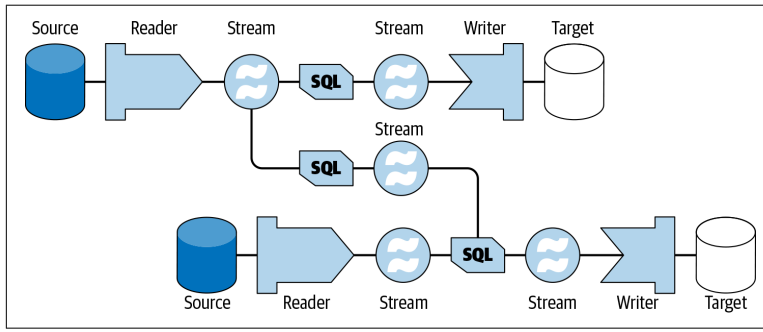


Figure 3-7. Performing processes using multiple steps

The rules and topologies discussed in the previous section also apply to these pipelines. Each of the streams in Figure 3-7 can have multiple implementations enabling single-threaded, multithreaded, multiprocess, and multinode processing, with or without partitioning and parallelization. Introduction of additional capabilities such as persistent streams, windowing, event storage, key/value stores, and caching add further complications to the physical implementations of data pipelines.

Stream-processing platforms need to handle deployment of arbitrarily complex data pipelines atomically (i.e., the whole pipeline is deployed or nothing is), adopting sensible default stream implementations based on partitioning, parallelism, resource usage, and other metrics while still allowing users to specify certain behaviors to optimize flows in production environments, as scale dictates.

Persistent Streams

As mentioned earlier, a data stream is an unbounded, continuous sequence of events in which each event comprises data and metadata (including timestamp) fields from external or intermediary data sources. Traditionally, to continuously run processing queries over streams, stream publishers and consumers used a classic publish/

subscribe model in which main memory was used to bound a portion of streaming data. This bound portion – either a single event or multiple events – was then examined for processing purposes and subsequently discarded so as to not exhaust main memory. Streams as such were always transient in nature. As soon as events in a stream were discarded, they could no longer be accessed.

There are several challenges that naturally arise when streams are processed in a purely in-memory manner, as just described:

- A subscriber must deal with streams as they are arriving. The consumption model is thus very tightly coupled to the publisher. If a publisher publishes an event, but the subscriber is not available – for example, due to a failure – the event could not be made available to the subscriber.

  If multiple data streams arrive into the stream-processing system, a subsequent replay of those streams from external systems cannot guarantee the exact order of previously acknowledged events if those events are discarded from memory.
- The publisher of the stream can stall if the consumer of the stream is slow to receive the stream. This has consequences on processing throughput.

Persistent streams are streams that are first reliably and efficiently written to disk prior to processing such that the order of the events is preserved to address the above challenges. This allows for the external source to first write the incoming stream's sequence of events onto disk, and for subscribers to consume those events independently of the publisher. The main thing is that this is transparent from an implementation standpoint.

CHAPTER 4
Stream Processing

The goal of stream processing is to get data immediately into the form that is needed for the target technology. That can involve any of the types of processing that we mentioned in Chapter 1, which we elaborate on in this chapter.

Although rare, there are use cases in which streaming integration is used to move data from a streaming source directly to a target without any in-stream processing. Here are examples of when this might occur:

- Replicating a database
- Moving changes from one database to another
- Reading from a message queue and writing the output as-is into a file
- Moving data from one filesystem to cloud storage without transforming the data

More commonly, however, the source data won't match the target data structure. This may be because some of the source data needs to be filtered out; for example, some events or fields of an event might not be needed, so they are removed. Or some data needs to be obfuscated because it contains personally identifiable information (PII). Perhaps additional fields need to be added before delivery to the target. Or, maybe the streaming data needs to be joined with some reference data for enrichment purposes. Stream processing

can perform all of these functions, continuously, with low latency, on any collected data (Figure 4-1).

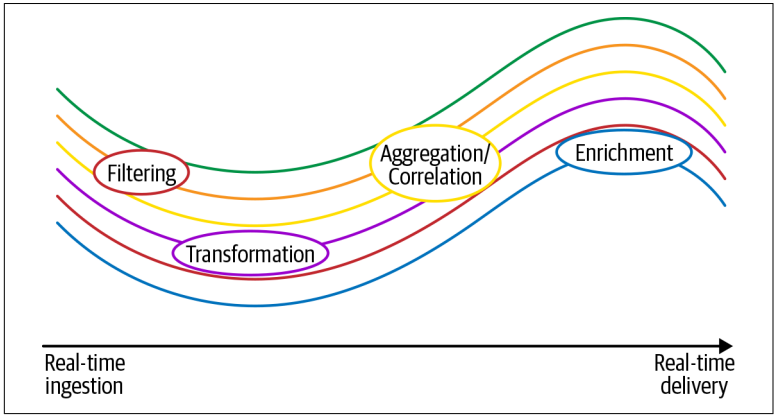


Figure 4-1. SQL-based stream processing with continuous queries

In-Memory

In a true streaming-integration platform, in-memory data processing is required. And that processing needs to be performed as efficiently as possible.

To achieve low latency and high throughput, it is critical to avoid writing data to disk or utilizing storage I/O prior to processing data. Stream processing needs to be performed directly on the streaming data in-memory, before the data ever lands on disk.

There are only two reasons to go to storage:

- The target being written to is a file-based system, such as a specific database or cloud storage.
- Persistent data streams are in use.

Stream processing also needs to be parallelized as necessary across multiple threads – or multiple processes and nodes – to achieve desired performance. Even in a multistage data pipeline, no disk I/O or writing data to storage should happen between the intermediate steps. All the processing should happen in-memory between receiving the data and writing the data into targets to achieve the desired throughput.

Continuous Queries

A streaming architecture also requires a modern querying paradigm. With database systems, queries are run against bounded sets of existing data. A single set of data is returned, and that's it. To see that query over time, you need to run the same query again – and again. To get updated results, you need to execute queries repeatedly.

With streaming systems, a single query is written based on the knowledge that data with a certain structure exists. That query sits in-memory, and waits for the data. As data appears on one or more incoming data streams, that query processes the incoming data and outputs results continuously in a never-ending fashion.

Effectively, there are two key differences between the in-memory continuous queries that happen in stream processing and the way people thought of queries in the past.

First, continuous queries work on a never-ending, infinite, and unbounded flow of data, as opposed to a bounded and known set of data that is resident in a table.

Second, although a database query is "one and done," continuous in-memory queries continually produce new results as new data is presented on the incoming data streams.

Unlike Extract, Transform, and Load (ETL) systems and integration technologies of the past, where things were batch-job oriented, real-time stream-processing systems run continuously, 24/7, and the engine behind the processing in those systems is the continuous query. Every time new records appear on a data stream, the query outputs new results.

It's important to understand that continuous queries aren't limited to simply reading from a data stream. They can read from in-memory caches, from in-memory reference data that might have been stored, or via windows (more on windows shortly). They can read from other – even persistent – storage, event and data sources, as well, depending on the architecture of the streaming system.

End-to-end latencies can range from microseconds to seconds, depending on how much processing is required, as opposed to the hours or even days typical of batch ETL solutions. As emphasized before, to achieve the goals of continually producing results and

producing them with very low latency, queries need to be in-memory.

SQL-Based Processing

There are many ways to process and manipulate data. We can do it via a scripting language, in an Excel spreadsheet, or even by writing lower-level code in Java, C++, Python or some other language.

In effect, there are three options in stream processing:

- Low-level code or APIs
- SQL-based processing
- UI-based building blocks that perform transformations at higher levels of definition

In our opinion, SQL is the best solution – a great compromise between the other two choices when you consider overall power, speed, and ease of use. We explain why in this chapter.

Consider the Users

First and foremost, the people who typically extract value from streaming data are data scientists, data analysts, or business analysts. They all have experience working with database systems, and almost all are very familiar with SQL as a data manipulation language.

By choosing SQL as your language for manipulating data, you allow the people who actually know the data to work with it firsthand, without intermediaries.

SQL is also very rich. It's easy to define filtering with *WHERE* clauses, to define column transformations, and to do conditional manipulations using case statements. Different types of objects can be *JOIN*ed as well as *GROUP BY*ed and aggregated. Whereas with databases, you're typically joining tables, in streaming cases, you're joining streams, windows, and caches to produce results. It's very easy to do that in SQL.

Of course, SQL is a higher-level *declarative language*. So, to achieve optimal performance, SQL must be transformed into high-performance code that executes on whatever stream processing platform has been chosen. If using Java, SQL is translated to

high-performance Java byte code. This way, you get the best of both worlds: the declarative nature of SQL – which allows data professionals to work directly with the data – and the high performance as if a developer had written the code.

Most streaming technologies are moving toward SQL for these reasons: Striim, Spark Streaming, Kafka, and Samsa, among others, offer SQL interfaces.

User Interface–Based Processing

Instead of providing a declarative language like SQL, some streaming vendors go to a higher level and do everything through a user interface (UI). This UI is typically a graphical user interface (GUI) and provides transformer components that are capable of doing some of the operations that SQL can do. Providing such an interface democratizes the use of data even more given that virtually any user can be proficient with a GUI.

Still, ultimately, what results is quite a lengthy data pipeline. That's because each of the GUI-based steps is performed as an individual task because each one of the transformers possesses very granular functionality. Whereas SQL could achieve its objective with a single statement – perhaps some filtering with a *WHERE* clause and some joins or column transformations – five or six different transformer boxes would need to be joined together using the GUI.

The upside of a GUI is that people who have no experience whatsoever in any programming language (including SQL) can build transformations. But there are downsides, as well. First, it might not be a good thing that people who possess no experience building out transformations are handling critical data. Second, the data pipelines themselves can suffer in terms of performance because, rather than a single processing step using an SQL statement, now quite a few processing steps are required. Performance can take a hit. Although having a GUI for the pipeline is essential, having multiple individual UI-based transformation steps is less efficient than a single SQL statement.

Multitemporality

As you recall, events are created whenever anything happens. If data is collected, an event is generated. If you're doing CDC from a database, or reading from files line by line, or receiving data from IoT devices or messaging systems, it is classified as an event. And each event has a timestamp for when it enters the system.

But it's also possible that there are additional time elements for such events. For example, with a database system, there is the time at which the event was committed into the database. Then there might be a timestamp for the time that the streaming system received it. Those two timestamps might be different, especially in a recovery scenario in which there is a discrepancy between when the database system wrote it and when it was read. Typically, there will be at least those two timestamps. They are considered *metadata*; that is, data about the data you've received.

These events will be at least bitemporal. But there could be additional timing elements within the data itself that are worth taking advantage of.

In an event system, you should be able to make use of any of those times for time-series analysis. You might do this to make sure the data is coming in the proper order, to use windowing functions and data aggregation in a time-based way, or to do time-series analyses using regression functions. The result is *multitemporality*, which means that any event can be defined to exist in time based on multiple pieces of time information.

This is important, because any operation that you perform on the data that relies on time should be able to make use of any of the timing attributes of an individual event. Simply utilizing the time at which it was collected might not be that useful. Being able to *choose* which of the timing elements can be better suited to your particular use case.

From an analytics perspective, however, timing information is crucial for building aggregates over time periods. We explain this later on when we explore windows and the different types of aggregations and functions that we can apply to data as part of time-series analyses.

Transformations

Transformations are about applying functions to incoming data. Transformations typically work on a record-by-record basis. The key is being able to manipulate the data, get it into the desired form, join it together, and perform functions on it to produce some desired output.

For example, you might want to concatenate two strings together so that you can combine first name and last name into a full name. That's a very simple example. Or, you might need to look up something based on an incoming value by saying, "Output the zip code that corresponds to this incoming IP address" by doing a *LOOKUP* function.

More complex functions are possible, of course, such as conditional transformations that involve case statements in SQL, in which if a particular field has a certain value, you want to combine it with a different field.

Filtering

Data flows in stream processing can be arbitrarily complex. For example, they could have splits or branches in them. Filtering is used when the output stream doesn't need all the data coming in.

Filtering for Data Reduction

Data reduction is one reason to do filtering. A simple example would be to avoid processing any debug log entries because you're interested only in warning or error messages. Another would be filtering incoming CDC records so that they don't include inputs from a particular database user. In the first case, a data element is being filtered. In the second case, the filter is based on metadata that includes which user made a given change because you don't want those particular changes downstream (Figure 4-2).

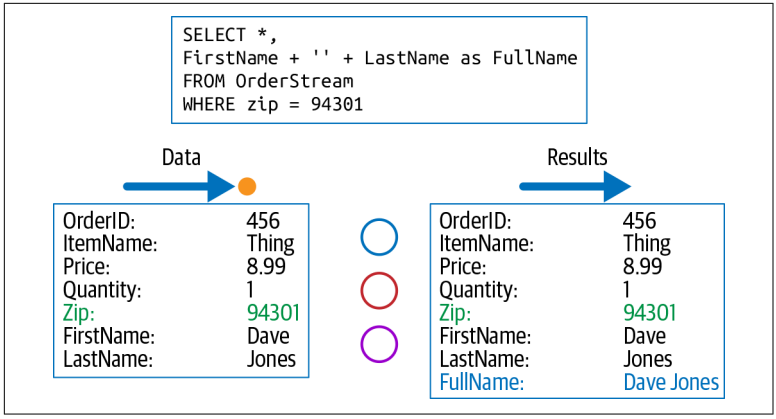


Figure 4-2. In-stream filtering is used when output streams do not require all incoming data

Filtering for Writing

Another reason to use filtering is to ensure that only certain data is written to certain targets. You might have an incoming data stream with a huge volume of data in it – a big database schema undergoing CDC, so that the incoming data stream includes changes from all the tables in that schema. But suppose that you want to store only information about product orders in a cloud data warehouse. You don't want changes to customer records or changes to products to be written to this particular reporting instance, just orders. Filtering enables this.

In SQL, most of the time, filtering is done by using the *WHERE* clause. In the cases of filtering based on aggregates, the *HAVING* clause is useful.

Analytics

We also can apply filtering for decision-making using analytics. You can use analytics to determine, for example, whether an event met or exceeded a specified threshold, or whether or not to generate an alert. We look deeper into analytics later on.

Windows

Windows are used to convert an infinite, unbounded incoming data stream into a finite bounded set of data, using whatever criteria is preferred (see Figure 4-3). Set-based operations can then be performed on that data. The two major uses of windows are *correlation* and *aggregation*. More on those later. There are several types of windows. *Sliding windows* change any time new data comes in, or as time passes. Every time a new record enters the window, or some time goes by, records can leave the window. Any queries that run on that sliding window are triggered every time the window changes.

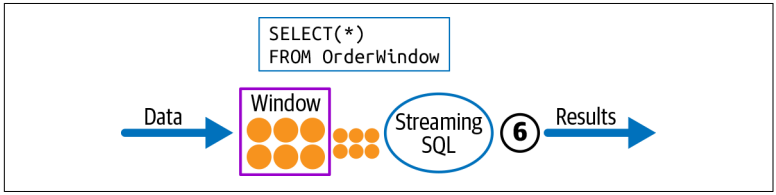


Figure 4-3. Windows are essential for correlation and aggregation use cases

Next, we have *jumping windows* or *batch windows*. These have criteria that determine how much data needs to be in the window before that data is output and before the queries reading from that window downstream are triggered with a windowful of data. The window is then emptied and ready to be filled again.

If you run a moving average over sliding windows, you see a smooth moving average rather than a jumping/batch window producing a moving average that happened only at a one-minute interval. Hybrid versions of that can be made, as well, where you say, "I don't want to execute the query every time I get new data, I want to do it every 10 events."

Then, there are *session windows*, which use timestamps. We could define such windows by, for example, "hold data until no new data corresponds to this for 30 seconds." This is useful when a customer comes to a website and is active for a period of time before leaving. Grouping all of their activities by waiting until they don't do anything else for a specified amount of time can trigger a query.

As such, there's an entire spectrum of sliding, to fully batch, and then session windows. And with any of the windows, we can also add timeouts that trigger the output independent of anything else happening. For example, "hold 100 events or output as much as happened in the last 30 seconds." You can do combinations of those windows, as well.

Windows are essential for correlation and aggregation use cases. The different types of windows are suited for different purposes. Yet windows might not be intuitive for database engineers. That's because databases are inherently set based. Data exists in a table, and that's it. Conceptualizing data as something that's continually changing, and having to create sets around it to do basic aggregate functions like sums, averages, or linear regressions, might be a new and different way of thinking.

Almost all streaming integration platforms offer some degree of windowing, and all are able to do timing-based window functions. However, not all support all the different types of windows or multi-temporality. This is important to know because windows are an essential component to any kind of any stream-processing platform. Without windows, streaming integration use cases can be limited.

Enrichment

Streaming data might not contain all of the information that you need at your target destination or to do analytics. That's when enrichment comes in.

For example, when doing CDC from a relational database, the majority of the fields in any one particular table will be IDs that refer to other tables. A data stream of all product items that have been ordered that is coming from the customer-order item table, for example, might contain an order ID, a customer ID, an item ID, maybe a quantity and timestamp, but that might be it.

Attempting to do downstream analytics on this limited data would probably not be productive. It would be impossible to write queries like, "show all of the real-time orders from California," or, "show all of the orders for umbrellas because a heavy storm is expected." Without additional information or additional content, you're not going to be able to perform rich analytics.

The answer is to enrich the existing data with reference data.

For example, products in an online store – let's say 100,000 of them – could be loaded into memory and indexed by ID. Then, whenever a customer order item appears in the data stream, it could be joined with the items in memory and additional information added: the item name, its category, its current inventory, and other relevant data. The data stream now has much more information in it, and is much more suited for analytics.

Distributed Caches

The challenge of enriching real-time data is the size and speed of the data. In a database, everything is in a data store. It's accessible within the same database. Two tables can easily be joined together to provide all the information needed. However, for real-time streaming environments, when we're talking about hundreds of thousands of events per second, this is difficult.

If you're joining against a remote database, for example, it would be necessary to do a query every event. Each query could take several milliseconds. With hundreds of thousands of events, it becomes impossible to query back to a database for every entry in a data stream in the required time. Similarly, with external caches or external data grids, it's not feasible to do a remote request from that cache and maintain that speed of 100,000 events per second.

We can deal with this by including a distributed cache, or in-memory data grid, within the streaming integration platform itself. By placing the data in memory into the same process space as the streaming data and partitioning that data in the same way as the incoming data events, it's possible to achieve very high throughput and low latency.

That doesn't always happen naturally. For example, with 100,000 items in memory, a six-node cluster, and a caching system defined to always maintain two copies of the data for redundancy purposes, the chances of any one particular item of data being on a single node is one in three.

However, if we can arrange it so that the events are partitioned by the same algorithm as is used for partitioning the reference data, the event will always land on the correct node. Now the query is completely in-memory in that node, and is really, really fast. Otherwise,

it's necessary to do a remote lookup, which could take tens to hundreds of microseconds.

Correlation

Correlation in this context does not refer to statistical correlation. It's not about matching variables or using linear regression to understand how variables are correlated. That's part of analytics (Chapter 5). Here, by correlation, we mean matching events coming in a data stream with events that are coming from one or more other data streams.

A simple example is to have data that represents activity on a number of different hosts, with that data coming from different sources. Perhaps it includes machine information, CPU usage, and memory coming from system logs. Maybe it includes network traffic information coming from network routers, or firewall information coming from other sources. How do you join it all together to see everything that's happening with a particular device?

In this case, they'd have an IP address or MAC ID in common. What's required then is to join the data streams together to produce a single output data stream.

However, doing this with data streams is difficult because they are so fast moving. Having events coinciding at exactly the same time is unusual. It's like aiming two proton beams at each other in a particle accelerator. The chances of two protons hitting are small because they're fast and they're tiny. The same is true for streaming events.

To join data streams together, you typically need to incorporate data windows. Imagine you have multiple physical pipelines and each has a temperature, rate of flow, and pressure. There are sensors on each pipeline measuring these attributes sending data to data streams, and each sensor is generating data at a different rate.

To understand the temperature, pressure, and flow of a particular pipe, it would be necessary to join those three data streams together. Now, because they all come at different speeds, the way to do that would be to create windows that have the last record per pipeline, per data stream. And whenever a new entry comes into the window, it would replace the old one for that pipeline.

The query is then written against the three windows. The query outputs an event whenever any of the windows change, and the output will be whatever the new value is for that pipeline on the window that changed, plus the existing measurements from the other windows.

This way, it's possible to join streams together that run at different speeds and produce an output whenever data is received on any one of the streams.

It's possible to go further than that by deciding to hold the last few values rather than only the last value. This allows a calculation to be made of what a value *could* be. Perhaps instead of simply using the last value, the average of the last 3 values is used, or a more complex regression mechanism could calculate the value based on the last 10 values.

In summary, windows aren't just useful for joining together streams that are coming at the same rate. It's also useful for joining streams that flow at different rates. Windows are the essential ingredient to being able to do real-time correlation across fast-moving data streams.

CHAPTER 5
Streaming Analytics

Analytics is an end goal of many streaming integration use cases. You can perform in a cloud data warehouse or by using machine learning on a large-scale data store. You can do it using an on-premises Hadoop infrastructure, or in a cloud storage environment. You could utilize a document store or another store like Azure Cosmos DB or Google Cloud Spanner. It could even be done writing into a database.

The most important point is that people want their data to be always up to date. So, when you're analyzing data, you should always possess the most recent data. And a primary driver of streaming analytics is that people want to do analytics on much more current data than was previously possible.

With ETL systems, people were satisfied with data that was a few hours or even a day old because they were running end-of-day reports, and that was the data that they wanted to see. With streaming systems, they want insight into the most current data. This is true whether the data is being analyzed in memory or is landing somewhere else.

However, getting real-time insights from data is typically not possible if the data needs to land somewhere (Figure 5-1). It's not possible to get within a few seconds – much less a subsecond from changes happening in the source system to being delivered into a target system that way. And it's still going to be necessary to trigger the analytics in that target platform somehow. Perhaps you're pulling it, or maybe you're running analytics reports, but you still must trigger it.

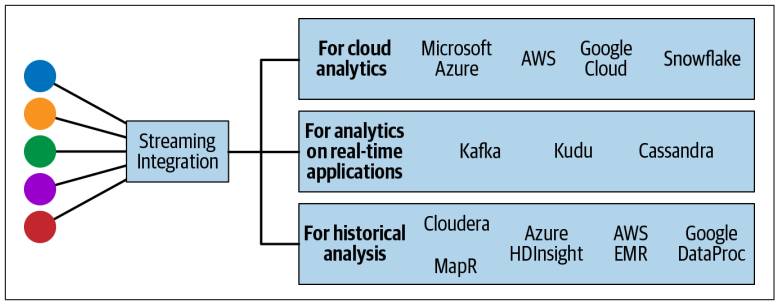


Figure 5-1. Streaming integration enables real-time analytics for cloud, applications, and historical data

In streaming analytics, the incoming data itself in the data stream is what triggers the analytics because it's continuously happening. If the goal is to have immediate notifications of anomalies, immediate insight into what's happening within data, or immediate alerts for unusual behavior, then streaming analytics is essential.

In this chapter, we discuss the most important aspects of streaming analytics, and how to get the most out of your data on your streaming platform.

Aggregation

Aggregation is any process in which information is gathered and expressed in a summary form. Because a data stream by definition is unbounded and infinite, doing aggregations over data streams is challenging. Suppose that you want to know the count and sum of a particular value in an accounts data stream. Both of those numbers are going to keep on increasing infinitely because data in the stream will just keep on coming. It is generally more useful to aggregate functions over a *bounded set* of that data.

Going back to the example of the order item stream, you might want to determine the top 10 selling items in an ecommerce store. In a database, that query is easy. You select the sum of the quantity, group by item ID, order by the sum of the quantity limit 10, and you have your top 10 selling items.

To change that query to know what sold most in the past five minutes, it would be necessary to put some limiters over timestamps. That query would need to be rerun whenever you wanted to see that value. In a streaming system, you can do time-constrained

queries much more easily by utilizing windows, as discussed in Chapter 4.)

To get an answer using this particular example, it would be necessary to create a window that holds five minutes' worth of data from the order item stream, and group by the item ID. Whenever anything changed, whenever any new data came into that window, that aggregate query would be rerun and it would show the sums all of the quantity sold per item in the last five minutes.

The advantage is that it's no longer necessary to keep running that query and changing the date premises. Everything is automated. That's why streaming analytics systems are much better suited to any analysis that is based on time. Streaming analytics is the optimal solution for time-series analysis.

Being able to group streaming data by some factor, aggregate over it, have it continually change, and have an output every time it changes, is key to aggregation. And it is also key to the summarization and analytics capabilities of streaming analytics. There are many different ways that we can do this, depending on the use case. It's even possible to have intermediate storage where the results of an aggregation can be stored in another window, and then you query that window.

On a practical level, with aggregation, it often makes sense to work backward and reverse engineer what actions to take based on the desired outcome. For example, if you want a dashboard visualization that shows in real time the top 10 items being sold based on changes in the underlying database table, you'd typically use that end result to determine which queries need to be written.

Going further with that example, now that you have visibility into top-selling items every five minutes, it might make sense to store those aggregates in another window. By storing the last hour's worth on a five-minute basis, it's possible to do additional queries over that. For example, you might want to be alerted if an item increases or decreases in sales by an anomalous amount within any five-minute period.

Pattern Matching

Pattern matching used to fall into a separate market niche called *complex event processing*. The purpose of complex event processing was to look at numerous small-grained business events and understand what was happening based on a pattern in those events (see Figure 5-2). With pattern matching, you're looking for data in a sequence of events from one or more data sources that corresponds to some particular pattern.

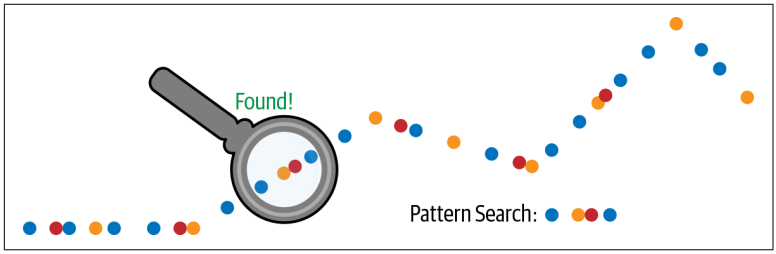


Figure 5-2. Understanding business events using pattern matching

For example, sensor information from an Internet of Things (IoT) device could include the temperature, pressure, flow rate, and vibration. If the temperature rose 10 degrees in that device, that might be within safety parameters. If the flow rate slowed, or the pressure increased, by a certain amount, that might also be within guidelines. However, it might be an indicator of trouble if the temperature rose by 10 degrees, the pressure went up by 10% and the flow decreased by 5% – all within a 10-second period.

By being able to look across those different sources and defining that pattern, an alert could be triggered.

Now the system isn't simply looking at one event or even the aggregation of events. It's looking at a sequence of events that meet exact criteria. We can define these happenings ahead of time, and then the data streams can be fed through the pattern matching. They will then output a result whenever the pattern matches.

Complex event processing is an essential part of streaming analytics and any streaming data platform must be able to do it to be considered a complete solution.

Statistical Analysis

Statistical analysis is when you apply statistical methods to real-time data. This is easy to do within a database table. For example, to calculate an average value, you simply select average column value.

However, generating a moving average by time over a database table is very difficult. Database tables aren't designed to do that. The query would need to be quite complex.

In a streaming system, doing statistical analysis on streaming data means utilizing the aggregate querying capability, but for statistical functions. We've already discussed aggregation; about being able to do a summary of values that were in a five-minute window. By replacing that sum with an average, now you have a five-minute average.

If, alternatively, you use a sliding window for which every time a new value comes to the window the output changes, the average now becomes a true real-time moving average. Similarly, you can do other statistical analyses.

Of course, certain things are impossible in real-time mode. The mean can be calculated, for example, but not the mode or the median. Those types of analyses don't work in a real-time data system. However, performing standard deviations or linear regressions certainly do work.

Imagine, in addition to doing a five-minute moving average, you're also doing a five-minute moving standard deviation. It's possible to check for a value that exceeds two times the standard deviation above the average or below the average, then an alert will be triggered because it's an anomalous value.

Thus, based on simple statistical analysis, it's possible to do interesting anomaly detections.

Integration with Machine Learning

Machine learning is a process by which computer systems can learn and improve from experience without being explicitly programmed. By inferring from patterns in data and generating algorithms and statistical models, computer systems can perform tasks without

being given explicit instructions. In short, they learn from the data that they are given.

What we've defined up until now are analyses that need to be specifically written. For example, you must specifically say, "this is the aggregation I want to perform," or "this is how I want to calculate these statistics and how I want to compare them." Or you must specifically say, "this is the exact pattern that I'm looking for, that if this pattern occurs, I want to trigger an alert."

A machine learning algorithm is different. The main thing about machine learning is that you don't necessarily know what you're looking for. The rules about what is "normal" are not known. It hasn't yet been determined what anomalous behavior means with regard to a particular data set. By integrating a trained machine learning model into a streaming analytics data flow, you would feed the current values to the model and then wait for results.

Alternatively, a model might be trained to understand the normal relationship between a set of variables in a data event. Then, by feeding it a set of variables, it can output what is normal versus what is unusual.

There's obviously a lot more that can be done using machine learning. Instead of simply having the two categories of normal and unusual, there could be different clusters that represent different types of behavior.

For example, if attempting to analyze a network, a machine learning algorithm could be trained to pick up on a number of different behaviors: normal user behavior, normal machine behavior, virus behavior, an external breach, or an external hack type of behavior. By classifying these behaviors, the machine learning model could trigger an alert for any events that fit into those categories.

The difference between streaming integration with machine learning versus other approaches is that machine learning is best suited for when you don't know what to look for in the data. You simply don't know how to write the rules.

The biggest challenge with integrating machine learning into a streaming environment – also known as *operationalizing* machine learning – is the way machine learning has traditionally worked. Historically, data professionals such as data analysts or data scientists were given a large volume of raw data. They would then spend,

on average, 80% of their time preparing that data: cleansing it, manipulating its structure, enriching it, and labeling it.

In other words, they would perform a lot of data manipulation ahead of time on the raw data to get it into the appropriate form for training. They would then train the model using a sample of that data. Only after all that time and effort would they have a well-trained model that represented the sample data.

Ideally, they should be able to give that model back to IT so that it can be run against the real-time data that's coming in.

But there's a problem. The model was trained on the prepared data that had various features extracted and was cleansed, enriched, and filtered. A lot of different tasks were performed to process that data before the model was trained. As such, the raw data that IT has might look nothing like the processed data used to train the model. There's a mismatch. A machine-learning model has been created, but it doesn't match the raw streaming data that is to be used to make predictions or spot anomalies.

The solution is to move as much of that data preparation as possible into the streaming system. The streaming system should be used for data cleansing, preparation, feature extraction, enrichment – for all of the tasks that the data scientist was previously doing *after* they received the data. This would now all be performed in the streaming system *beforehand*. That prepared data then can be utilized to train the machine learning model. The benefit is minimizing latency. The historical way to train the machine learning model is based on data that is out of date because it takes the data scientists so much time to prepare it and apply it to the model. In a streaming architecture, the data is prepared in milliseconds so that it is still current data.

If desired, there can be a simultaneous process during which the streaming system is still writing the training files, but it's also passing the real-time streaming data into the machine learning algorithm so that it can return real-time results. And those real-time results could be classified into different categories of data. It could be making predictions into the future or looking at the difference between a predicted value and an actual value. But it's doing this based on machine learning training, not on any hardcoded values.

Anomaly Detection and Prediction

Anomaly detection and prediction are the end goals of streaming analytics. If unusual behavior is identified – perhaps unusual network behavior, unusual sales of a particular item, or the temperature of a device increasing while pressure also rises – that behavior could indicate a potential problem that might require an alert. This is one of the top benefits of streaming analytics: alerts on critical issues based on complex calculations that can be done in real time. With such real-time alerting, it's possible to know immediately whether you have a network breach, you've mispriced your flat-screen televisions, or that there's a problem in a manufacturing pipeline.

Unlike other analytic systems in which queries are made after the fact to understand what has happened, streaming analytics systems can automatically send immediate notifications, without human intervention.

Alerting based on anomalies, pattern matching, and statistical analyses are all key aspects of streaming integration. We can extend these functions to make predictions. In addition to generating alerts immediately, you can also utilize visualizations or dashboards to see predicted outcomes based on what's going on right now. This can help with quality assurance, customer experience, or other business concerns.

It can also help with predictive maintenance. For example, based on real-time information, we can identify that a particular motor is likely to wear out within the next two weeks, as opposed to its expected full-year lifespan, because data from it has been matched against machine learning analyses that have been done on failing motors in the past.

You can use predictive analytics to identify the current state of a sensor in a factory, a pipeline, or a fleet of cars, and utilize that information to make decisions.

Being able to not only identify anomalies, but also make predictions based on all the streaming data that you have and present that effectively to users – which we talk about in Chapter 6 – is the primary goal of streaming analytics.

CHAPTER 6
Data Delivery and Visualization

Chapter 2 discusses collecting real-time data, and the different techniques that you can use to obtain streaming data from different sources. This chapter looks at the opposite of that: now that you have your streaming data, where do you put it?

Generally, when starting to work with streaming data, businesses have a specific problem – or problems – to solve. This means the target, or where the data will be placed, is top of mind from the beginning. Which target you choose and the way that target will solve the specific problems you face will be very different depending on the particular use case.

Certain technologies are better suited for real-time applications. They're superior at gaining insights from data as it flows in. Others are better suited for long-term storage and then, downstream, to perform large-scale analytics. Still others, instead of delivering data into some ultimate location, make it possible to get immediate insights by performing processing or analytics within the streaming integration platform.

In this chapter, we talk about streaming data targets: databases, files, Hadoop or other big data technologies, and messaging systems, among others. We also investigate the prevalence of cloud services and how moving to the cloud can alter the way you use target technologies. This latter point is very important given that the majority of businesses have already begun their transition to the cloud.

Here are some of the questions we answer about working with streaming data targets:

- How do you determine the end goal of your streaming integration?
- Where should you put streaming data?
- What should you do with it?

The first step when choosing a target technology is to understand your use case. For example, a common goal in a streaming initiative is to migrate (or replicate) an on-premises database to (into) a cloud database. This can be because an application is being moved from on-premises infrastructure to the cloud. Or perhaps some reporting on an on-premises application is being done in the cloud.

In most of these cases, a database is the optimal target – a database that's similar to the original database and kept synchronized from an on-premises source to a database target. However, if the goal is to do long-term analytics, or perhaps even machine learning, on data that was gathered in a continuous fashion using streaming technologies, a database might not be the most appropriate target.

Another option would be to move the data into storage. This could be done using a Hadoop technology like Hadoop Distributed File System (HDFS), or you could put it into a cloud storage environment like Amazon Simple Storage Service (Amazon S3), Microsoft Azure Data Lake Store, or Google Cloud Storage. Your target could be a cloud data warehouse or even an on-premises data warehouse. Again, it depends on what you want to get out of the streaming data – what your particular use case, or goal, is.

This is why, when looking at streaming integration platforms, an important consideration is their ability to write data continuously into your technology of choice and to be able to single source the data once from wherever you're pulling it from, whether a database, messaging system, or IoT device.

After you articulate your goal, it's time to consider the target. We can repurpose data in multiple ways. Some of it can be written to an on-premises messaging system like Kafka. Some can go into cloud storage. And some can go into a cloud data warehouse. The important point is that a streaming integration platform should be able to work with all of the different types of target technologies. Data that's

been collected once can also be written with different processing technologies to multiple different targets.

The techniques to work with these different targets are as varied as they are for data collection (see Chapter 2). Which one to choose also depends on the mode and the nature of the source data as well as its mission criticality, which we discuss more fully in Chapter 7. But it's important to keep mission criticality in mind from the beginning because you must determine ahead of time, based on your use case, how important the streaming data in question is. Can you afford to lose any of it? What would happen if you had duplicate data? Based on that you can determine the level of reliability that is needed and choose your target given that some targets are better suited to highly reliable scenarios than others.

In this chapter we discuss each of the different potential targets and their respective pros and cons.

Databases and Data Warehouses

When you write into databases from a streaming integration system, you need to consider a number of issues:

- How do I want to write the data?
- How do I deal with different tables?
- How do I map the data from a source schema to a target schema?
- How do I manage data-type differences from, for example, an Oracle database to a Teradata database?
- How do I optimize performance?
- How do I deal with changes?
- How do I deal with transactionality?

These questions persist across all the variants of databases and data storage that you might need to integrate with as a target, both on-premises and in the cloud. It's also important to include document stores, MongoDB, and newer scalable databases like Google Spanner and Azure Cosmos DB that are available only in the cloud.

Whenever writing into any of these databases, for example, the first step is to identify how to translate the data. This means taking the

event streams from the streaming integration platform and transforming them into what you want them to appear like in the database. This typically requires a lot of configuration. Whatever streaming integration platform you choose must be capable of enabling you to do this.

To configure your data streams correctly, you must consider your use case. For example, Java Database Connectivity (JDBC) is the most common way to connect when a database is the target on a Java-based streaming processing platform. But even with that relatively simple technology, you need to make configuration choices based upon the specific use case.

If you're collecting real-time IoT data, and the goal is to insert all of the data into a table in a target database, you could write it as inserts into that table. That's a very simple use case.

However, recall from Chapter 2 that one of the major techniques for collecting data from databases is CDC, or being able to collect database operations, inserts, updates, and deletes as changes. When doing that, it might be necessary for certain use cases to maintain a change – to keep track of whether it was an insert as opposed to an update or a delete – and to keep that information about the data available so you can take appropriate action in the target database.

Take migrating an on-premises database into the cloud. Not only is a snapshot in time of all of the data in that database required, it's important to recognize that unless you can stop the application related to that database from operating, changes are going to continue to be made to that database. You cannot stop most mission-critical applications. They're too important to the business.

To move an important application to the cloud, then, requires also moving dynamically changing data in the database to the cloud. This is referred to as either an *online migration* or *phased migration* to the cloud. Simply making a copy of whatever is in the database at a given moment doesn't work.

Here's why. An initial copy of the data can be captured at a particular point in time, but then any changes that happen to that data from that point until you are ready to start using that new database must be added to the target. Only then can it be tested to ensure that the application is working and subsequently rolled into production.

In this use case, you would effectively utilize a combination of batch-oriented load of data from tables into a target database, but then overlay that with changes that have been – and which continue to be – made to that data. This provides you with the ability to not just copy an on-premises database, but to synchronize that database to a new cloud instance.

The point is, it's reasonably trivial to write all data in a streaming architecture to a target database as inserts. It's more complex to honor the changes (e.g., to apply inserts into a target as inserts, updates as updates, and deletes as deletes). It becomes even more complicated to honor transactionality, and transaction boundaries that might have been present on the source.

For example, if five changes happened within a single transaction on a streaming source, it might be necessary to ensure that those five changes – perhaps a couple of inserts, an update, and a delete – make it all the way to the target database and that they are done within the scope of a single transaction. That's because they need to either all be there or not.

Therefore, for databases, you need to be able to handle very high throughput, respect change, and honor transaction boundaries. When batching operations together, and considering parallelizing operations for performance reasons, you need to do this while keeping transactional integrity and potential relationships within the database tables in mind.

Data warehouses have their own challenges. The biggest difference between writing streaming data into databases and writing it into data warehouses is that the latter typically are not built to receive streaming data one event at a time. They are more likely designed to be bulk loaded with large amounts of data in one go, and that doesn't always align with the event-by-event nature of streaming integration. Thus, when considering on-premises data warehouses, all of them work better when you can load them in bulk.

The same is true of cloud data warehouses. Amazon Redshift, Google BigQuery, Azure SQL Data Warehouse, and Snowflake all operate better from a loading perspective if you do it in bulk. But even with data warehouses, loading bulk inserts isn't always the answer. Most of these data warehouses support modifications through different mechanisms that can be quite convoluted, especially when

you aren't loading all the data, but trying to apply changes that might have occurred.

When using streaming integration, it is possible to keep a cloud data warehouse like Snowflake synchronized with an on-premises relational database like Oracle by utilizing the proper technology integration points. You must keep in mind, however, that real-time latencies aren't usually possible when writing into a data warehouse. Data warehouses are typically not designed to receive data in a real-time fashion, and they certainly are not designed to also create the kind of data structures typically found in a database in real time from source data. Typically, there is a batch-processing step that needs to happen, which means that the best kind of latencies possible with data warehouses, even with streaming integration technologies, are in the order of minutes, not seconds or subseconds.

Files

Files are really nothing more than chunks of data that have been named. It's very common to write streaming data into file targets, on-premises and in the cloud. In this section, we talk about both. We're also going to discuss files separately from technologies like HDFS, which is part of the Hadoop ecosystem, for reasons that will be clear in the next section.

When writing to files, it's important to consider what data is going into them. Is all of the data being stored in one big file? Is it a "rolling" file so that as soon as it reaches a certain size or triggers some other criteria, it rolls over to a new file? That scenario would, of course, require you to implement a numbering or naming system to distinguish the files from one another.

Perhaps the goal is to write to multiple files in parallel based on some criteria in the data. For example, if you're reading change data from databases, you might want to split it based on metadata such as timestamps or source location and write it into one file per table.

Because of this, the ability of your streaming platform to take streaming data and choose which of potentially multiple buckets it goes into is an essential consideration. Here are some of the questions that arise:

- How do you split the data?
- How do you write the files?
- How do you name the files?
- Where do they reside (on-premises or in the cloud)?
- Are they managed by a storage system, or are they directory entries?
- What is the format of the data?

These are all important considerations, but the last, the format of the data, is the most critical. When writing data to files, what does it look like? Does it look like raw data? Perhaps it is delimited in some way. Maybe it needs to be in JSON, which is structured human-readable text. Or perhaps one of the industry-standard binary formats.

Then there's the decision of which technology to use to work with the filesystem. Is it Hadoop? Is it cloud storage? All of those considerations are very much based upon the use case and the end goal. For example, are the files going to be used to store things for long-term retrieval? Are they going to be utilized for doing machine learning somewhere down the line? The important point is that the streaming data platform you choose is flexible enough to accommodate what you want to achieve, across a wide variety of use cases.

Storage Technologies

The technology used to store data as files has changed dramatically in the past few years. It has transitioned from simple directory structures on network filesystems, through distributed scalable storage designed for parallel processing of data in the on-premises Hadoop big data world, to almost infinite, elastically scalable and globally distributed storage using cloud storage mechanisms.

Within the Hadoop world, the HDFS is the basic technology. However, you might also want to write into HBase as a key/value store, or write data using Hive – a SQL-based HDFS interface – into a filesystem. You might even be using a filesystem as an intermediary for some other integration that ends up being transparent to the user.

For example, if writing data into the Snowflake database, it might be desirable to land intermediary files on Amazon S3 or Azure Data

Lake Store, which can then be picked up immediately by Snowflake and loaded as external tables. That storage system can act as a staging area for loading into HDFS, in which case all the questions we discussed about how often to write and the format of the data still must be answered. This might or might not be abstracted from the user perspective, depending on which streaming integration system is being used.

Typically, with Hadoop-based systems, external tables for HDFS play a large role in where the data is targeted. Kudu, a newer, real-time data warehouse infrastructure built on top of Hadoop does this. Although working with raw data can be important, it might be necessary to abstract it so that you can write directly into a Hadoop-based technology and make it easier to process or analyze.

There are many blurred lines between these various storage technologies, and trade-offs are often required. When uploading to the cloud, the interfaces don't always provide ways of writing real-time streams but require file uploads, so determining an optimal micro-batch strategy that optimizes speed, latency and cost is important.

Messaging Systems

Messaging targets are very important as targets because they continue the good work of stream processing that have already been started by the streaming integration platform.

Just as it is important to be able to read from messaging systems (as is discussed in Chapter 2), it's important to be able to write into them. Your reasons for using messaging systems as targets will vary considerably by use case. It might be that processed or analyzed data is being delivered for other downstream purposes, or that the messaging system is being utilized as a bridge between different environments or different locations.

Messaging systems can be useful as a way to enable others to work with the same data or as an entry point into a cloud environment. And because they are inherently streaming themselves, messaging systems can often be the jumping-off point to something else while maintaining a streaming architecture. For example, a use case might require taking real-time database information or real-time legacy data and pushing that out into a cloud messaging system like Azure

Event Hubs. Then, that real-time change data becomes the backbone for future processing or analytics.

When working with messaging systems, it's important to understand the major differences in both the APIs used to access them, and the underlying capabilities of the messaging system that determine what can and can't be done using that messaging system as a target.

In addition to popular on-site messaging systems such as IBM MQ, Java Message Service (JMS)–based systems, and Kafka, it's also important to consider newer cloud native technologies, and technologies that span on-premises and cloud such as Solace. Cloud native systems include Amazon Kinesis, Google Pub/Sub, and Azure Event Hubs. These, however, require specific APIs to work with them.

JMS offers a common way for Java applications to access any JMS-compliant system. With JMS-based messaging systems, there are both queues and topics that can be written into. Both can have transactional support as well persistent messaging, ensuring that everything can be guaranteed to be written at least once into JMS targets.

However, for cases in which exactly-once processing is required, queues can be challenging. Topics can have multiple subscribers for the same data, so it's possible, such as in the case of restart after failure, to query a topic to determine what has been written. Queues are typically peer to peer and thus cannot be queried in the same way and need different strategies to ensure exactly-once processing. We discuss this more in Chapter 7.

On the other hand, if you use a Kafka-based messaging system, every reader of a topic is independent, and maintains its own index of what has been written, allowing you to get around this challenge. Although Kafka is very popular, it's not the only messaging system out there, and streaming integration platforms that work only with Kafka are extremely limiting.

JMS-based messaging systems give you the ability to integrate with a lot of different messaging systems through a single technique. Alternatively, generic protocols like Advanced Message Queuing Protocol (AMQP) allow you to work with many different providers. Which approach you choose depends on your specific use case.

Application Programming Interfaces

It may also be important to deliver real-time data to a target destination through application programming interfaces (APIs). This isn't a new subject because all of the target technologies we've discussed also work through APIs. JMS has an API, Kafka has an API, databases have APIs, and at the lowest level, you have APIs to write to files, as well.

But when we talk about APIs in this section, we're focusing on business application–level APIs. The working definition of API we're using is: an interface or communication protocol that dictates how two systems can interact. These can be RESTful APIs or streaming socket–based APIs. But they have the same goal: to deliver real-time streaming data directly into an application.

For example, the Salesforce API would provide the ability to write into Salesforce and update records continually using streaming data. APIs can be used to deliver data continuously to everything from a mission-critical enterprise application, to a project-management support system, to a bug-tracking system, to keep them up to date. The suitability of any particular API for delivery from streaming integration depends a lot on workload and the API mechanism. RESTful APIs which require a request/response per payload might not scale to high-throughput rates, unless they are called with microbatches of data. Streaming APIs such as WebSockets can scale more easily but often require more robust connection management to avoid timeouts and handle failures.

Cloud Technologies

Working with cloud technologies is essential for any streaming integration platform because so many streaming use cases involve moving data continuously in real time from an on-site application to the cloud.

From a streaming integration perspective, cloud technologies aren't that different to working with their on-premises counterparts, including databases, files, and messaging systems. What *is* different is how you connect into the cloud.

It's important to understand that each cloud vendor has its own security mechanisms. These are proprietary security rules that you

must follow, and they include proprietary ways to generate the keys needed to connect. Each cloud vendor also has its own secure access control. Therefore, in addition to thinking about the target technology itself – the database, the file system, the messaging system – you need to consider how you use that technology to access whatever cloud vendor(s) you select (Figure 6-1).

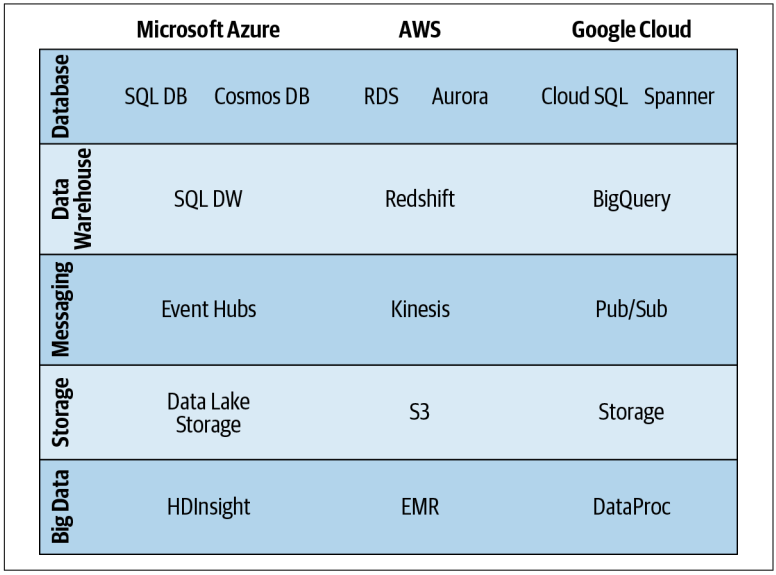


Figure 6-1. A streaming platform should support different technologies across cloud vendors

Streaming integration platforms should support all of this. Databases, data warehouses, storage, messaging systems, Hadoop infrastructures, and other target technologies should be able to connect to and across Google Cloud, Amazon AWS, and Microsoft Azure. And, significantly, not only should your streaming platform support all these different technologies across all these different clouds, they must support multiple instances of them across multiple cloud platforms.

Here are some considerations when the cloud is your target for streaming data integration:

- What cloud platform should you choose (or which one[s] have already been chosen at your organization)?
- Who manages the cloud?

- Am I allowed to access it directly, or do I need to use an internally defined proxy?
- How do I access it?
- What authentication do I need?
- Where do I store credentials to get in?

Many quite complicated cloud use cases exist. For example, some digital transformation use cases involve two-way moving of data from on premises up to the cloud and then back down from the cloud to on premises. Some involve multicloud scenarios with single-source data, some of which is going into one cloud technology hosted by one cloud provider, others going into another cloud technology hosted by another cloud provider.

You even have intercloud use cases. That's when the streaming integration platform is used to synchronize a database provided by one cloud provider, say AWS, into a database on another cloud provider, like Google. The reason you might do this is to have a totally redundant backup that's continually updated and hosted by a different cloud provider for peace of mind.

In summary, delivery into cloud by definition involves working both with the target technology you choose, but also working with a particular cloud or clouds. Much depends on the source of the data and the purpose of the integration. Keep in mind, additionally, that all of these different technologies to choose from as targets could also be used as sources from the cloud, as discussed in Chapter 2.

Visualization

When attempting to manage all these data pipelines, streaming integrations, data processing, and data analytics you have put into place, it is natural to want to know: what is happening? You might want to simply do basic monitoring, but more frequently you will have particular SLAs or key performance indicators (KPIs) that you need to attain. You might want to trigger alerts if analytics identifies that certain thresholds have been breached. In these cases, you want some sort of visualization.

Visualization in streaming integration is a little different to what most people are accustomed to in BI applications. This is because the data being visualized is moving in real time. You're seeing

up-to-the-second results of the analytics that you've done. You're seeing what's happening right now.

So, visualization for real-time data must support continuous and rapidly changing data. Your streaming platform needs to support mechanisms for displaying this data: be capable of producing different charts, graphs, tables, and other options for viewing the data. Perhaps you are tracking multiple KPIs or SLAs. Maybe you want to track alerts or anomalies detected. You might want to look at trend lines and see the direction in which your data is heading.

It might also be important that you can drill down from a high-level view to a more detailed view. This could be based on analytics to get more insight into real-time data. Being able to dynamically filter data is critical when it comes to real-time visualization. And this should be easy to do; your users should have as easy a time working with real-time streaming data through an integration platform as they do working with large amounts of static data through standard business intelligence tools.

CHAPTER 7
Mission Criticality

When it comes to streaming integration – or simply any type of real-time data processing – it's fairly straightforward to build the integrations we've been talking about in a lab scenario. You can try it, piece it together with bits of open source, test it, and generally get things working as a viable proof of concept (POC).

But, when it comes time to put these things into production, that's another matter altogether. When building continuous data pipelines that need to run around the clock and support the applications that the business depends upon, then many questions arise:

- How do I make sure that it works 24/7?
- How do I ensure that it can handle failure?
- How can I be sure that it can scale to the levels I need it to, given the expected growth of data within my organization?
- How do I make sure that the data remains secure as it moves around the enterprise at high speeds?

This is all about *mission criticality*. And it can't be an afterthought. You need to design your streaming integration infrastructure with mission criticality in mind. This is one of the biggest issues to consider when investigating streaming integration platforms.

Clustering

We've talked about the benefits of building a distributed platform for scaling streaming integration. Clustering[1] is one of the most popular methods for building a distributed environment. But there are several ways to build a scalable and reliable cluster. Your choice of stream-processing platform becomes critical at this point. When doing your due diligence, make sure that the platform has the following characteristics:

- Requires little upfront configuration
- Is self-managing and self-governing
- Allows users to add and remove cluster nodes without interrupting processing flow

The old-school way of scaling was to have a standalone node that could be scaled up by adding hardware. You would simply add more CPUs and more memory to that standalone machine. Of course, for reliability reasons, you would have replicated the application and its data to a backup node and switched over to it in case of failure. That's how most enterprises structured their databases.

Early incarnations of stream processing used the same paradigm. Because distributed architecture had not yet reached mainstream, the pioneers of stream processing relied on scaling on top of a single machine. But there are limitations to this approach. Among other things, memory management in a Java-based environment causes performance to suffer if you take the single-processor route.

However, with the new breed of stream processing, we use a modern method, one made popular by the Hadoop approach to big data. This paradigm involves building large, scalable clusters of nodes over which the stream integration and processing can be distributed.

Because of this, it's important that the stream integration platform you choose supports clustering. It should also be able to handle scalability, failover, and recovery by distributing the load or failing over

1 Clustering refers here to a set of nodes that are ready to run one or more streaming pipeline. There are no jobs in streaming, because they run continually. After the pipelines are deployed to the cluster they run until stopped.

the load to another node in the cluster if one or more machines go down.

Your streaming integration platform must also make all of this easy to work with. Some platforms are very configuration heavy, especially in the Hadoop world, and you must do an immense amount of configuration upfront to determine which nodes in a cluster should take on which tasks and what purpose each node serves. That's why we recommend choosing a platform that has a "low-touch" way of building out a cluster.

The low-touch approach basically allows the software to determine what's best. Effectively, you start up a certain number of nodes, those nodes talk among themselves, and they automatically determine which nodes services should run on, including management services, monitoring services, and all the required other services to keep stream integration and processing humming along nicely. Additionally, a low-touch stream integration platform allows you to add or remove nodes without interrupting processing.

Scalability and Performance

Scalability is obviously an essential aspect of building a successful stream integration system. As we discussed in the previous section, you need to do this very differently from the traditional ways that enterprises have approached scalability. There's only so much you can scale on a single node.

So scaling out rather than scaling up is best (Figure 7-1). But as we explained, your stream integration platform should be able to add additional nodes to the cluster as necessary, as the load increases. Additionally, the cluster should be self-aware, aware of the resources it's using on every node, and be able to distribute the load and scaling out as necessary to maintain performance.

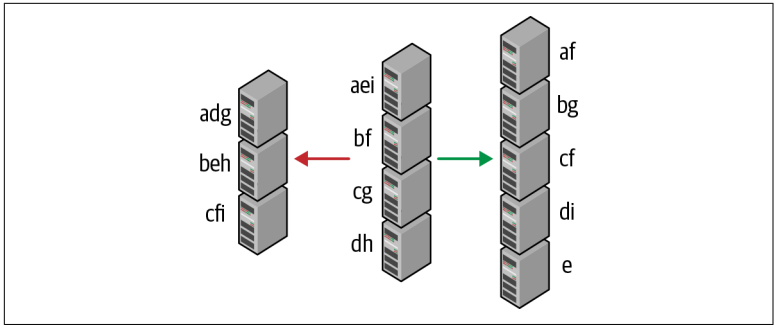


Figure 7-1. Scaling out using clustering

Keep in mind that different parts of the architecture can require different scale-out capabilities. It might be possible to read incoming data at a certain rate. For example, one part might be reading changes that are happening within a database at a rate of 100,000 changes per second. But, downstream from that, some processing might be necessary, and maybe that processing is quite complex, and the fastest that processing task can run is 25,000 changes per second. In such a case, you have a problem.

If a distributed clustered streaming integration platform is designed well, it should be possible to partition the events that are being processed over the available nodes. For example, if there are four nodes in the cluster, the streaming integration platform should be able to partition the processing over those nodes, with each one processing 25,000 changes per second. Overall, that downstream processing is occurring at 100,000 per second, keeping up with your reader.

Not only should you be able to parallelize events and have different pieces of that streaming architecture running in different volumes around the cluster, you should also be able to dictate which pieces scale. For example, you might not be able to scale the reader, because it will work only if it has one connection into your source. Perhaps you're reading from a database change log, or you're reading from a messaging queue that can have only one reader to guarantee the order of events. In those cases, you don't want to scale the reader, but you might want to scale the processing. Make sure that the streaming platform you choose allows you to do this.

To give another example, in our experience, the biggest bottleneck to a distributed clustered architecture like this is the writing into the various targets (Chapter 6). You might want to have a single reader,

maybe some parallelism across the cluster for the processing, but you might want even more parallelism in the writing side, for the data targets. In such cases, it's crucial that you can scale not just the cluster, but the individual streaming components that are running within the cluster.

This brings us to an important point to make about scalability: it is irrevocably tied to performance. The example we just gave – the need to be able to scale individual streaming components within a cluster – is critical for performance.

Another important scalability-related performance concept is that you should attempt to do everything on a single node, as much as is possible. This includes reading from a source, all the processing, and the writing to a target. A single node should handle it all.

This is because every time data is moved between nodes, either on a single machine or on separate physical machines, data is moving over the network. This involves serialization and deserialization of the objects that you're working with, which inevitably adds overhead.

In environments that require high levels of security, that data might also be encrypted as it moves over the network. In such cases, you need to worry about network bandwidth and also the additional CPU usage required to serialize and deserialize, and encrypt and decrypt, that data.

From a performance perspective, when designing data flows, you want to try and keep those hops between physical instances of the cluster as low as possible. Ideally, you would read once, and if you need to distribute it out over a cluster for processing, you would do that once, but then it should stay on the same nodes as it passes down the data pipeline. Similarly, for performance reasons, you don't want to allow input/output (I/O) functions such as writing to disk between individual processing steps.

Enterprises struggle with streaming integration architectures because of these challenges. As a result, they end up with unnecessary overhead on their distributed environments. For example, if utilizing a persistent messaging system between every step in the data pipeline, it is physically writing over the network, serializing the data, maybe encrypting it, writing it to disk, then reading from

disk, and then deserializing it and possibly having to decrypt it, as well. That's a long pipeline.

That is going to add overhead, and in high-performance, scalable, low-latency systems, you want to minimize overhead. So as much as possible, try to avoid I/O, and stick to in-memory processing of data only.

Constant serialization and deserialization of data can also be a problem. Ideally, your streaming platform should make this easy to do. For example, if you're choosing to work with SQL for ease-of-use reasons, you still want to make sure that at runtime it is as fast as if you'd actually written custom code yourself. So, deploy a stream integration platform that gives you highly optimized code even if generated by SQL.

Another point related to scalability and performance: as we mentioned before, when working with distributed caches, it's important to make sure that if you're joining event data with data that you've loaded into memory in a distributed cache, for either reference or context purposes, you still can bring the event to the physical machine that actually possesses the data for that cache. That's because the cache itself can be distributed.

Some of the data will be present on only some of the nodes in the cluster. You want to make sure that each event lands on a node in the cluster that physically has the data that it's going to join with. Otherwise, a remote lookup will be necessary, invoking serialization and deserialization and significantly slowing performance.

In summary, when evaluating stream processing platforms, be cautious. A system might work well when testing in a lab with low throughput, when scalability isn't a factor. Before going into production, you will need to perform scalability tests to ensure that your distributed architecture will work – and stay low-latency – when processing high-volume loads.

Failover and Reliability

The next three topics we cover – failover, recovery, and exactly-once processing – are about reliability: how to ensure that you can recover data, events, and processing results in case of a system failure, large or small.

First, failover. The stream-processing platform you choose needs to recognize that failures can and will happen and must have ways of dealing with those failures.

There are many different types of failure. There are physical machine crashes, and there are software crashes. There are problems reading from data sources. There are difficulties writing into targets that are outside of the platform. There may be issues with the network, causing extended outages. The streaming integration platform needs to be able to be aware that these things happen. Your platform needs to be able to handle all of them effectively and with ease.

In the case of the failure of a single node in the cluster that's doing stream processing, you should be able to failover data flow to another node, or another set of nodes, and have it pick up where it left off, without losing anything.

This is where the cluster comes in with regard to reliability. For scalability, as you recall, the cluster is important because of parallelism. For reliability, it's important that you have *redundancy* so that you can easily have an application automatically failover to other nodes if the node that the application is running on goes down for some reason.

In scalability, we talked about adding nodes to scale. You also want to add nodes so that the cluster can handle nodes disappearing, and still maintain the application.

Now, operating in a cluster doesn't necessarily mean that the application will continue running uninterrupted. If all of the pieces of an application are running on a single node that goes down, by definition, it's going to be interrupted. The point of failover, however, it should be able to pick up from where it left off, and catch up, using the same recovery techniques that we discuss in the next section.

From an external perspective, there may be a slight bump in the processing rate, but after a period of time, things should go back to normal without losing any data.

Recovery

Recovery is about coming back from a failure with operations and data intact.

You might not necessarily care whether you lose some data in a failure. Completely recovering from a failure might not be a significant issue in such cases. Perhaps you're monitoring a real-time IoT device. Even if you miss some data while things are failing over, the results of the analysis aren't going to be affected. You just might have a short period of time in which you don't have insight into the device you are monitoring.

But in most cases – especially when the application is mission-critical and the results have to accurately represent all the incoming data – full recovery becomes very important.

There are quite a few recovery strategies employed by streaming integration platforms. For example, a platform can replicate states, so that for every operation that occurs on one node, the state required for that tool occurs on another node.

The issue is that it takes bandwidth and resources to do that replication. If you're doing it for every event that's coming in, that process is going to consume a lot of additional resources, which could considerably slow down the data flow. After all, replicating requires network transformation in the form of all the serialization and deserialization we've discussed. That will severely slow things down.

An alternate way of handling recovery is to enable checkpointing. Checkpointing means that you are periodically recording where you are in terms of the *processing*. What was the last event received? What was the last event output from the system before it went down?

Checkpointing takes into account the different types of constructs that can exist between a source and a target. That includes queries, windows, pattern matching, event buffers, and anything else that happens to the data. For example, you might be joining messages together, you might be discarding messages, or you might be aggregating certain pieces of data before you output it into a target. So, it's not necessarily a case of simply remembering the last event that came into the system.

There is a caveat: using checkpointing for recovery does require rewindable data sources. If you are recording states using checkpointing, perhaps while utilizing data windows, and you have a minute's worth of data in a window to recover, you need to rebuild that minute's worth of data, and then pick up outputting data after you get to the point where you're able to start processing again.

Effectively, a low "watermark" in checkpointing is a pointer into your data source that represents the first event from which you need to rebuild the state: the point of failover. A high watermark is the first new event that needs to be processed.

This means that if an entire cluster fails, you would need to be able to go back in time and ask your data sources, "please give me the last minute's worth of data again." And although it's possible to go back in time and reread data with certain types of data sources such as change data or file sources, with other types of data sources it simply won't work. Certain messaging systems, and especially IoT data, for example, are not rewindable. First, you might not be able to contact them in that way. Second, they might not record that amount of information themselves. Small IoT devices, for example, physically might not be able to replay the data.

That's where the persistent data streams that we talked about earlier in the book become important. They record all of the incoming data at the source so that when you are in a recovery situation, you can rewind into that data stream, pick up from the low watermark, and then continue processing when you get to the high watermark. When using persistent data streams, using checkpointing for recovery has a much lower impact on processing because you need only periodically record checkpoints.

Of course, even the mechanism where you replicate a state across a cluster could fail if the entire cluster disappeared. Some of the checkpointing could have issues in the place where you're recording where the checkpoint has disappeared.

One of the strategies to deal with that is to record checkpoints alongside a target. For example, say you are writing to a database as a target, and the data delivered are the results of stream processing. If you could write the checkpoint information also into that database, you could always pick up from where you left off. That's because the data is present with the data target that you're writing to. The information about how the target got into this state is also

present within the target. That's true for databases. It can also be true for messaging systems and some other types of targets, as well.

Exactly-Once Processing

Understanding recovery brings us to the different types of reliability guarantees that you can have in a streaming integration system. For the purposes of this book, there are *at-least-once processing* and *exactly-once processing*.

At-least-once processing is where you guarantee that every incoming message will be read and delivered to a target. But it doesn't guarantee that it will be delivered to a target only once.

Exactly-once processing is what it sounds like. It means that every incoming event is processed, but you write the results of that processing only once to the target. Once exactly. You do not write it again. You never have duplicates.

Why does this matter? Some recovery scenarios, trying to be extra cautious not to miss anything, might write data or events or other functions into a target where they have already been written. That works satisfactorily in some circumstances, either because the target can deal with it, or downstream processing from the target can handle it.

But there are other cases for which you do need exactly-once processing. This isn't possible all the time, but theoretically it is possible for situations in which you can write checkpointing information into the target. It's also possible where you can ask the target, "what was the last event that was given to you?" Then, based on that, the platform can determine the next data it should be given.

By validating the data that's been written to a target, by writing checkpoints to a target, or by including metadata in the data that you write to a target, you can ensure exactly-once processing.

Security

The final criteria around mission criticality is security (Figure 7-2). The three major issues about security when it comes to stream processing are as follows:

- Ensuring that you have policies around who can do what to the data
- Encrypting data in flight
- Encrypting data at rest

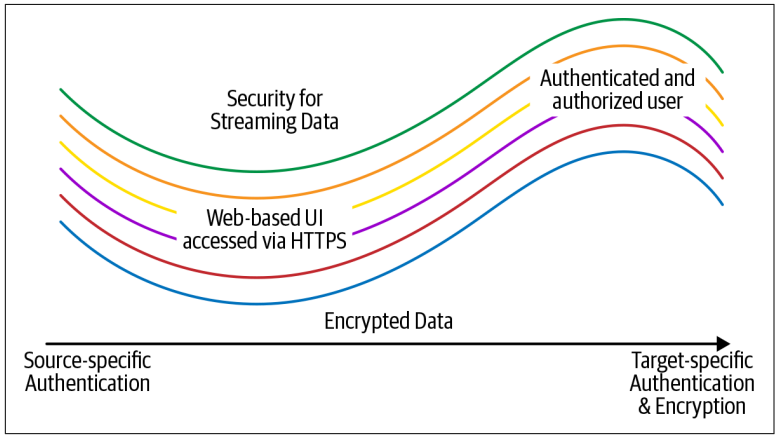


Figure 7-2. Various forms of security applied to streaming data

In this section, we'll discuss these three issues.

Access Control

The first aspect of security centers around who has access. That means the importance of being able to stipulate which individuals are authorized to work with data streams, who can start and stop applications, and who can view data but do nothing else.

For this type of security, you need to make sure that the stream-processing platform you're using has end-to-end, policy-based security. You need to be able to define different sets of users, and which kinds of permissions each set possesses.

With leading streaming-integration platforms that also offer analytics, you should also be able to specify which users can build data flows. Perhaps some users are restricted to only setting up initial data sources. Other users can't access those data sources but can work with preprocessed streams. Still other users can build dashboards and analytics. And finally, those who are often called "end" users, who are only authorized to view the end result of the analytics. They can simply work with the final dashboards, and that's it.

In summary, you can have developers, and you can have administrators and monitors of data. And then you can have end users who are looking at analytics results. Each has access to different aspects of the data.

Another part of security involves being able to identify intermediate raw data streams that might contain PII or other sensitive data. For example, a data flow might exist in which you obfuscate, or mask, some of that data. Because the data is going to be used for analytics, it's not necessary to have the real data there. In fact, having the raw data displayed could be a security issue.

In such scenarios, the streaming platform's authorization mechanism and policies kick in. You could make sure that people only were authorized to access the intermediate data streams, the ones that contained the obfuscated sensitive data.

These kinds of security policies, security restrictions, and user access rights need to be able to be applied across all of the components in a homogeneous way.

Encrypting Data in Flight

The other side of security revolves around data protection, which is ensuring that data is encrypted and thus not visible to unauthorized users. Because streaming platforms typically aren't persisting data – unless you're using a persistent data stream – and they're not writing to intermediate data stores, the most important place that data can be encrypted is within the data streams themselves, when they're moving from node to node.

They're going across a network, so it's essential that a stream-processing platform permits you to encrypt the data as it's flowing across a network, in an industry-standard way, based on what your company policies are.

But also, if you are utilizing persistent data streams that do write data to disk, those persistent streams should also be encrypted, as well, almost by default, because you are writing it to disk.

Thus, encrypting data in stream, as it's moving, or if it's being stored in a persistent data stream, is essential.

Encrypting Data at Rest

When writing data to targets, your streaming platform should have the ability to encrypt data based on the capabilities of the targets. It should be able to provide the correct configuration and credentials to enable whatever encryption is available with that target. However, for the security-conscious organization, encrypting data with its own keys and algorithms, and managing keys security should also be possible through the streaming-integration platform.

Additionally, typically sources and targets often contain important information that must remain secret. A database might possess a password that is needed to connect to it. Or a cloud service might have an access key to allow connection. Your stream processing platform must protect that information by encrypting it in an industry-standard way so that no one can hack in and acquire credentials that give them access to sensitive systems.

These represent the three most critical considerations about security and streaming platforms. The ability to address security within your streaming integration solution is key to supporting mission-critical applications.

CHAPTER 8
Conclusion

We are entering a new era – the era of real-time data. This data is driving a vast digital transformation, where even the most conservative enterprises are modernizing through technology and the data it provides. The goal? To elevate the customer experience, optimize processes, and outperform the competition. This is happening on a global scale. We called this *data modernization*, and streaming integration is its engine, providing the way to realize value from this modernization and from the real-time data fueling it.

In this book, we described how to achieve data modernization through streaming integration in detail to help you understand how it can be applied to solve the very real business challenges you face in a world transformed by digital technologies. We started with a history of data, before introducing and defining the idea of streaming integration and why it is so critical to businesses today.

We saw how streaming integration can solve disparate use cases, by driving hybrid cloud adoption and providing a platform for real-time applications while also being the engine delivering continuous data for a variety of reporting and analytics purposes. Whether an organization is performing a zero-downtime data migration to a cloud database; continually updating a data warehouse; collecting, aggregating and reacting to IoT data, or performing real-time machine learning predictions, streaming integration can allow businesses to reap immediate benefits.

Building streaming data pipelines is the first important step. Then, stream processing – getting the data into the form that is needed

through filtering, transformation, enrichment, and correlation – becomes critical before advancing to streaming analytics. We wrapped up the book talking about data delivery and visualization, and, finally, about the mission-criticality of data.

Throughout this book, we discussed what organizations should expect of an enterprise-grade streaming integration platform to realize the value in their real-time data. Such a platform needs, at a minimum, to offer continuous real-time data ingestion from a broad range of heterogeneous sources, high-speed in-flight stream processing, and sub-second delivery of data to both cloud and on-premises endpoints in the correct format to be immediately available to their high-value operational workloads.

It is essential that such a platform is also highly scalable, reliable, and secure, and provides an intuitive GUI interface to facilitate transformation of real-time stream processing into an organizational asset that delivers both bottom- and top-line benefits to the enterprise.

By utilizing a streaming integration platform that operates in-memory, provides CDC and other ingestion capabilities, and works in today's distributed hybrid technology environments, enterprises can perform data modernization initiatives that were impossible using batch or legacy integration techniques.

Streaming integration is truly the engine of digital transformation.

About the Authors

Steve Wilkes is a life-long technologist, architect, and hands-on development executive. Prior to founding Striim, Steve was the senior director of the Advanced Technology Group at GoldenGate Software. Here he focused on data integration, and continued this role following the acquisition by Oracle, where he also took the lead for Oracle's cloud data integration strategy. His earlier career included Senior Enterprise Architect at The Middleware Company, principal technologist at AltoWeb and a number of product development and consulting roles including Cap Gemini's Advanced Technology Group. Steve has handled every role in the software lifecycle and most roles in a technology company at some point during his career. He still codes in multiple languages, often at the same time. Steve holds a Master of Engineering Degree in microelectronics and software engineering from the University of Newcastle-upon-Tyne in the UK.

Alok Pareek is a founder of Striim and head of products. Prior to Striim, Alok served as Vice President at Oracle in the Server Technologies development organization, where he was responsibile for product strategy, management, and vision for data integration, and data replication products. Alok also led the engineering and performance teams that collaborated on architecture, solutions, and future product functionality with global strategic customers. Alok was the Vice President of Technology at GoldenGate where he led the technology vision and strategy from 2004 through its acquisition by Oracle in 2009. He started his career as an engineer in Oracle's kernel development team where he worked on redo generation, recovery, and high-speed data movement for over ten years. He has multiple patents, has published several papers, and has presented at numerous academic and industry conferences. Alok has a graduate degree in Computer Science from Stanford University.