O'REILLY®

Stream Processing with Apache Flink

Fundamentals, Implementation, and Operation of Streaming Applications

Grayscale Edition

For Sale in the Indian Subcontinent & Select Countries Only*

*Refer Back Cover

SPD®

Fabian Hueske &
Vasiliki Kalavri

Stream Processing with Apache Flink

Fundamentals, Implementation, and Operation of Streaming Applications

Fabian Hueske and Vasiliki Kalavri

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

SPD®

SHROFF PUBLISHERS & DISTRIBUTORS PVT. LTD.

Mumbai Bangalore Kolkata New Delhi

Stream Processing with Apache Flink

by Fabian Hueske and Vasiliki Kalavri

 ISBN: 978-1-491-97429-2
Originally 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://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Rachel Roumeliotis
Development Editor: Alicia Young
Production Editor: Katherine Tozer
Copyeditor: Christina Edwards
Proofreader: Charles Roumeliotis

Indexer: Judith McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

April 2019: First Edition

Revision History for the First Edition: 2019-04-03: First Release
See http://oreilly.com/catalog/errata.csp?isbn=9781491974292 for release details.

First Indian Reprint: April 2019

ISBN: 978-93-5213-828-9

Published by **Shroff Publishers and Distributors Pvt. Ltd.** B-103, Railway Commercial Complex, Sector 3, Sanpada (E), Navi Mumbai 400705 • TEL: (91 22) 4158 4158 • FAX: (91 22) 4158 4141• E-mail : spdorders@shroffpublishers.com
Web : www.shroffpublishers.com Printed at Jasmine Art Printers Pvt. Ltd., Mumbai

Table of Contents

Preface

What You Will Learn in This Book

This book will teach you everything you need to know about stream processing with Apache Flink. It consists of 11 chapters that hopefully tell a coherent story. While some chapters are descriptive and aim to introduce high-level design concepts, others are more hands-on and contain many code examples.

While we intended for the book to be read in chapter order when we were writing it, readers familiar with a chapter's content might want to skip it. Others more interested in writing Flink code right away might want to read the practical chapters first. In the following, we briefly describe the contents of each chapter, so you can directly jump to those chapters that interest you most.

- Chapter 1 gives an overview of stateful stream processing, data processing application architectures, application designs, and the benefits of stream processing over traditional approaches. It also gives you a brief look at what it is like to run your first streaming application on a local Flink instance.
- Chapter 2 discusses the fundamental concepts and challenges of stream processing, independent of Flink.
- Chapter 3 describes Flink's system architecture and internals. It discusses distributed architecture, time and state handling in streaming applications, and Flink's fault-tolerance mechanisms.
- Chapter 4 explains how to set up an environment to develop and debug Flink applications.
- Chapter 5 introduces you to the basics of the Flink's DataStream API. You will learn how to implement a DataStream application and which stream transformations, functions, and data types are supported.

- Chapter 6 discusses the time-based operators of the DataStream API. This includes window operators and time-based joins as well as process functions that provide the most flexibility when dealing with time in streaming applications.
- Chapter 7 explains how to implement stateful functions and discusses everything around this topic, such as the performance, robustness, and evolution of stateful functions. It also shows how to use Flink's queryable state.
- Chapter 8 presents Flink's most commonly used source and sink connectors. It discusses Flink's approach to end-to-end application consistency and how to implement custom connectors to ingest data from and emit data to external systems.
- Chapter 9 discusses how to set up and configure Flink clusters in various environments.
- Chapter 10 covers operation, monitoring, and maintenance of streaming applications that run 24/7.
- Finally, Chapter 11 contains resources you can use to ask questions, attend Flink-related events, and learn how Flink is currently being used.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords. Also used for module and package names, and to show commands or other text that should be typed literally by the user and the output of commands.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element signifies a warning or caution.

Using Code Examples

Supplemental material (code examples in Java and Scala) is available for download at *https://github.com/streaming-with-flink*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Stream Processing with Apache Flink* by Fabian Hueske and Vasiliki Kalavri (O'Reilly). Copyright 2019 Fabian Hueske and Vasiliki Kalavri, 978-1-491-97429-2."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For almost 40 years, *O'Reilly* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text

and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/stream-proc*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Follow the authors on Twitter: *@fhueske* and *@vkalavri*

Acknowledgments

This book couldn't have been possible without the help and support of several amazing people. We would like to thank and acknowledge some of them here.

This book summarizes knowledge obtained through years of design, development, and testing performed by the Apache Flink community at large. We are grateful to everyone who has contributed to Flink through code, documentation, reviews, bug reports, feature requests, mailing list discussions, trainings, conference talks, meetup organization, and other activities.

Special thanks go to our fellow Flink committers: Alan Gates, Aljoscha Krettek, Andra Lungu, ChengXiang Li, Chesnay Schepler, Chiwan Park, Daniel Warneke, Dawid Wysakowicz, Gary Yao, Greg Hogan, Gyula Fóra, Henry Saputra, Jamie Grier, Jark Wu, Jincheng Sun, Konstantinos Kloudas, Kostas Tzoumas, Kurt Young, Márton Balassi, Matthias J. Sax, Maximilian Michels, Nico Kruber, Paris Carbone, Robert

Metzger, Sebastian Schelter, Shaoxuan Wang, Shuyi Chen, Stefan Richter, Stephan Ewen, Theodore Vasiloudis, Thomas Weise, Till Rohrmann, Timo Walther, Tzu-Li (Gordon) Tai, Ufuk Celebi, Xiaogang Shi, Xiaowei Jiang, Xingcan Cui. With this book, we hope to reach developers, engineers, and streaming enthusiasts around the world and grow the Flink community even larger.

We've also like to thank our technical reviewers who made countless valuable suggestions helping us to improve the presentation of the content. Thank you, Adam Kawa, Aljoscha Krettek, Kenneth Knowles, Lea Giordano, Matthias J. Sax, Stephan Ewen, Ted Malaska, and Tyler Akidau.

Finally, we say a big thank you to all the people at O'Reilly who accompanied us on our two and a half year long journey and helped us to push this project over the finish line. Thank you, Alicia Young, Colleen Lobner, Christine Edwards, Katherine Tozer, Marie Beaugureau, and Tim McGovern.

CHAPTER 1

Introduction to Stateful Stream Processing

Apache Flink is a distributed stream processor with intuitive and expressive APIs to implement stateful stream processing applications. It efficiently runs such applications at large scale in a fault-tolerant manner. Flink joined the Apache Software Foundation as an incubating project in April 2014 and became a top-level project in January 2015. Since its beginning, Flink has had a very active and continuously growing community of users and contributors. To date, more than five hundred individuals have contributed to Flink, and it has evolved into one of the most sophisticated open source stream processing engines as proven by its widespread adoption. Flink powers large-scale, business-critical applications in many companies and enterprises across different industries and around the globe.

Stream processing technology is becoming more and more popular with companies big and small because it provides superior solutions for many established use cases such as data analytics, ETL, and transactional applications, but also facilitates novel applications, software architectures, and business opportunities. In this chapter, we discuss why stateful stream processing is becoming so popular and assess its potential. We start by reviewing conventional data application architectures and point out their limitations. Next, we introduce application designs based on stateful stream processing that exhibit many interesting characteristics and benefits over traditional approaches. Finally, we briefly discuss the evolution of open source stream processors and help you run a streaming application on a local Flink instance.

Traditional Data Infrastructures

Data and data processing have been omnipresent in businesses for many decades. Over the years the collection and usage of data has grown consistently, and companies have designed and built infrastructures to manage that data. The traditional architecture that most businesses implement distinguishes two types of data process-

ing: transactional processing and analytical processing. In this section, we discuss both types and how they manage and process data.

Transactional Processing

Companies use all kinds of applications for their day-to-day business activities, such as enterprise resource planning (ERP) systems, customer relationship management (CRM) software, and web-based applications. These systems are typically designed with separate tiers for data processing (the application itself) and data storage (a transactional database system) as shown in Figure 1-1.

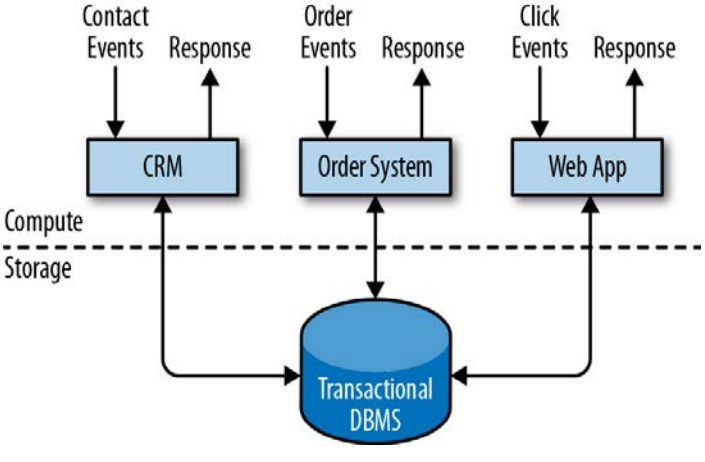

Figure 1-1. Traditional design of transactional applications that store data in a remote database system

Applications are usually connected to external services or face human users and continuously process incoming events such as orders, email, or clicks on a website. When an event is processed, an application reads its state or updates it by running transactions against the remote database system. Often, a database system serves multiple applications that sometimes access the same databases or tables.

This application design can cause problems when applications need to evolve or scale. Since multiple applications might work on the same data representation or share the same infrastructure, changing the schema of a table or scaling a database system requires careful planning and a lot of effort. A recent approach to overcoming the tight bundling of applications is the microservices design pattern. Microservices are designed as small, self-contained, and independent applications. They follow the UNIX philosophy of doing a single thing and doing it well. More complex applications are built by connecting several microservices with each other that only communicate over standardized interfaces such as RESTful HTTP connections. Because

microservices are strictly decoupled from each other and only communicate over well-defined interfaces, each microservice can be implemented with a different technology stack including a programming language, libraries, and datastores. Microservices and all the required software and services are typically bundled and deployed in independent containers. Figure 1-2 depicts a microservices architecture.

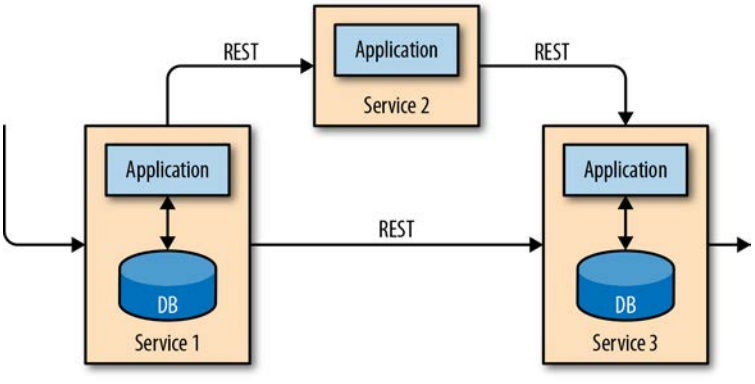

Figure 1-2. A microservices architecture

Analytical Processing

The data that is stored in the various transactional database systems of a company can provide valuable insights about a company's business operations. For example, the data of an order processing system can be analyzed to obtain sales growth over time, to identify reasons for delayed shipments, or to predict future sales in order to adjust the inventory. However, transactional data is often distributed across several disconnected database systems and is more valuable when it can be jointly analyzed. Moreover, the data often needs to be transformed into a common format.

Instead of running analytical queries directly on the transactional databases, the data is typically replicated to a data warehouse, a dedicated datastore for analytical query workloads. In order to populate a data warehouse, the data managed by the transactional database systems needs to be copied to it. The process of copying data to the data warehouse is called extract–transform–load (ETL). An ETL process extracts data from a transactional database, transforms it into a common representation that might include validation, value normalization, encoding, deduplication, and schema transformation, and finally loads it into the analytical database. ETL processes can be quite complex and often require technically sophisticated solutions to meet performance requirements. ETL processes need to run periodically to keep the data in the data warehouse synchronized.

Once the data has been imported into the data warehouse it can be queried and analyzed. Typically, there are two classes of queries executed on a data warehouse. The

first type are periodic report queries that compute business-relevant statistics such as revenue, user growth, or production output. These metrics are assembled into reports that help the management to assess the business's overall health. The second type are ad-hoc queries that aim to provide answers to specific questions and support business-critical decisions, for example a query to collect revenue numbers and spending on radio commercials to evaluate the effectiveness of a marketing campaign. Both kinds of queries are executed by a data warehouse in a batch processing fashion, as shown in Figure 1-3.

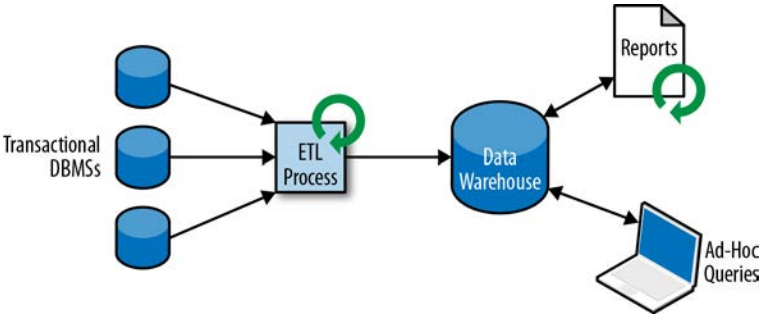

Figure 1-3. A traditional data warehouse architecture for data analytics

Today, components of the Apache Hadoop ecosystem are integral parts in the IT infrastructures of many enterprises. Instead of inserting all data into a relational database system, significant amounts of data, such as log files, social media, or web click logs, are written into Hadoop's distributed filesystem (HDFS), S3, or other bulk datastores, like Apache HBase, which provide massive storage capacity at a small cost. Data that resides in such storage systems can be queried with and processed by a SQL-on-Hadoop engine, for example Apache Hive, Apache Drill, or Apache Impala. However, the infrastructure remains basically the same as a traditional data warehouse architecture.

Stateful Stream Processing

Virtually all data is created as continuous streams of events. Think of user interactions on websites or in mobile apps, placements of orders, server logs, or sensor measurements; all of these are streams of events. In fact, it is difficult to find examples of finite, complete datasets that are generated all at once. Stateful stream processing is an application design pattern for processing unbounded streams of events and is applicable to many different use cases in the IT infrastructure of a company. Before we discuss its use cases, we briefly explain how stateful stream processing works.

Any application that processes a stream of events and does not just perform trivial record-at-a-time transformations needs to be stateful, with the ability to store and

access intermediate data. When an application receives an event, it can perform arbitrary computations that involve reading data from or writing data to the state. In principle, state can be stored and accessed in many different places including program variables, local files, or embedded or external databases.

Apache Flink stores the application state locally in memory or in an embedded database. Since Flink is a distributed system, the local state needs to be protected against failures to avoid data loss in case of application or machine failure. Flink guarantees this by periodically writing a consistent checkpoint of the application state to a remote and durable storage. State, state consistency, and Flink's checkpointing mechanism will be discussed in more detail in the following chapters, but, for now, Figure 1-4 shows a stateful streaming Flink application.

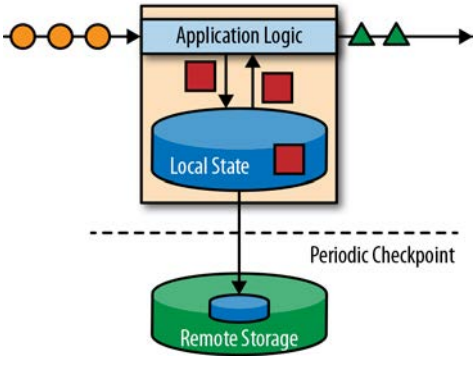

Figure 1-4. A stateful streaming application

Stateful stream processing applications often ingest their incoming events from an event log. An event log stores and distributes event streams. Events are written to a durable, append-only log, which means that the order of written events cannot be changed. A stream that is written to an event log can be read many times by the same or different consumers. Due to the append-only property of the log, events are always published to all consumers in exactly the same order. There are several event log systems available as open source software, Apache Kafka being the most popular, or as integrated services offered by cloud computing providers.

Connecting a stateful streaming application running on Flink and an event log is interesting for multiple reasons. In this architecture the event log persists the input events and can replay them in deterministic order. In case of a failure, Flink recovers a stateful streaming application by restoring its state from a previous checkpoint and resetting the read position on the event log. The application will replay (and fast forward) the input events from the event log until it reaches the tail of the stream. This technique is used to recover from failures but can also be leveraged to update an

application, fix bugs and repair previously emitted results, migrate an application to a different cluster, or perform A/B tests with different application versions.

As previously stated, stateful stream processing is a versatile and flexible design architecture that can be used for many different use cases. In the following, we present three classes of applications that are commonly implemented using stateful stream processing: (1) event-driven applications, (2) data pipeline applications, and (3) data analytics applications.

Real-World Streaming Use-Cases and Deployments

If you are interested in learning more about real-world use cases and deployments, check out Apache Flink's Powered By page and the talk recordings and slide decks of Flink Forward presentations.

We describe the classes of applications as distinct patterns to emphasize the versatility of stateful stream processing, but most real-world applications share the properties of more than one class.

Event-Driven Applications

Event-driven applications are stateful streaming applications that ingest event streams and process the events with application-specific business logic. Depending on the business logic, an event-driven application can trigger actions such as sending an alert or an email or write events to an outgoing event stream to be consumed by another event-driven application.

Typical use cases for event-driven applications include:

- Real-time recommendations (e.g., for recommending products while customers browse a retailer's website)
- Pattern detection or complex event processing (e.g., for fraud detection in credit card transactions)
- Anomaly detection (e.g., to detect attempts to intrude a computer network)

Event-driven applications are an evolution of microservices. They communicate via event logs instead of REST calls and hold application data as local state instead of writing it to and reading it from an external datastore, such as a relational database or key-value store. Figure 1-5 shows a service architecture composed of event-driven streaming applications.

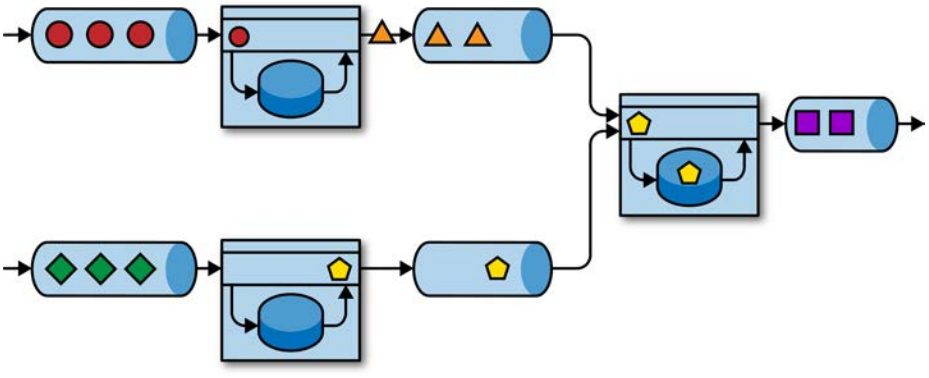

Figure 1-5. An event-driven application architecture

The applications in Figure 1-5 are connected by event logs. One application emits its output to an event log and another application consumes the events the other application emitted. The event log decouples senders and receivers and provides asynchronous, nonblocking event transfer. Each application can be stateful and can locally manage its own state without accessing external datastores. Applications can also be individually operated and scaled.

Event-driven applications offer several benefits compared to transactional applications or microservices. Local state access provides very good performance compared to reading and writing queries against remote datastores. Scaling and fault tolerance are handled by the stream processor, and by leveraging an event log as the input source the complete input of an application is reliably stored and can be deterministically replayed. Furthermore, Flink can reset the state of an application to a previous savepoint, making it possible to evolve or rescale an application without losing its state.

Event-driven applications have quite high requirements on the stream processor that runs them. Not all stream processors are equally well-suited to run event-driven applications. The expressiveness of the API and the quality of state handling and event-time support determine the business logic that can be implemented and executed. This aspect depends on the APIs of the stream processor, what kinds of state primitives it provides, and the quality of its support for event-time processing. Moreover, exactly-once state consistency and the ability to scale an application are fundamental requirements for event-driven applications. Apache Flink checks all these boxes and is a very good choice to run this class of applications.

Data Pipelines

Today's IT architectures include many different datastores, such as relational and special-purpose database systems, event logs, distributed filesystems, in-memory caches, and search indexes. All of these systems store data in different formats and data structures that provide the best performance for their specific access pattern. It is common that companies store the same data in multiple different systems to improve the performance of data accesses. For example, information for a product that is offered in a webshop can be stored in a transactional database, a web cache, and a search index. Due to this replication of data, the data stores must be kept in sync.

A traditional approach to synchronize data in different storage systems is periodic ETL jobs. However, they do not meet the latency requirements for many of today's use cases. An alternative is to use an event log to distribute updates. The updates are written to and distributed by the event log. Consumers of the log incorporate the updates into the affected data stores. Depending on the use case, the transferred data may need to be normalized, enriched with external data, or aggregated before it is ingested by the target data store.

Ingesting, transforming, and inserting data with low latency is another common use case for stateful stream processing applications. This type of application is called a data pipeline. Data pipelines must be able to process large amounts of data in a short time. A stream processor that operates a data pipeline should also feature many source and sink connectors to read data from and write data to various storage systems. Again, Flink does all of this.

Streaming Analytics

ETL jobs periodically import data into a datastore and the data is processed by ad-hoc or scheduled queries. This is batch processing regardless of whether the architecture is based on a data warehouse or components of the Hadoop ecosystem. While periodically loading data into a data analysis system has been the state of the art for many years, it adds considerable latency to the analytics pipeline.

Depending on the scheduling intervals it may take hours or days until a data point is included in a report. To some extent, the latency can be reduced by importing data into the datastore with a data pipeline application. However, even with continuous ETL there will always be a delay until an event is processed by a query. While this kind of delay may have been acceptable in the past, applications today must be able to collect data in real-time and immediately act on it (e.g., by adjusting to changing conditions in a mobile game or by personalizing user experiences for an online retailer).

Instead of waiting to be periodically triggered, a streaming analytics application continuously ingests streams of events and updates its result by incorporating the latest events with low latency. This is similar to the maintenance techniques database sys-

tems use to update materialized views. Typically, streaming applications store their result in an external data store that supports efficient updates, such as a database or key-value store. The live updated results of a streaming analytics application can be used to power dashboard applications as shown in Figure 1-6.

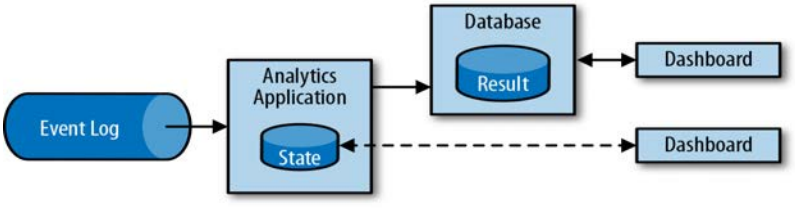

Figure 1-6. A streaming analytics application

Besides the much shorter time needed for an event to be incorporated into an analytics result, there is another, less obvious, advantage of streaming analytics applications. Traditional analytics pipelines consist of several individual components such as an ETL process, a storage system, and in the case of a Hadoop-based environment, a data processor and scheduler to trigger jobs or queries. In contrast, a stream processor that runs a stateful streaming application takes care of all these processing steps, including event ingestion, continuous computation including state maintenance, and updating the results. Moreover, the stream processor can recover from failures with exactly-once state consistency guarantees and can adjust the compute resources of an application. Stream processors like Flink also support event-time processing to produce correct and deterministic results and the ability to process large amounts of data in little time.

Streaming analytics applications are commonly used for:

- Monitoring the quality of cellphone networks
- Analyzing user behavior in mobile applications
- Ad-hoc analysis of live data in consumer technology

Although we don't cover it here, Flink also provides support for analytical SQL queries over streams.

The Evolution of Open Source Stream Processing

Data stream processing is not a novel technology. Some of the first research prototypes and commercial products date back to the late 1990s. However, the growing adoption of stream processing technology in the recent past has been driven to a large extent by the availability of mature open source stream processors. Today, distributed open source stream processors power business-critical applications in many

enterprises across different industries such as (online) retail, social media, telecommunication, gaming, and banking. Open source software is a major driver of this trend, mainly due to two reasons:

1. Open source stream processing software is a commodity that everybody can evaluate and use.
2. Scalable stream processing technology is rapidly maturing and evolving due to the efforts of many open source communities.

The Apache Software Foundation alone is the home of more than a dozen projects related to stream processing. New distributed stream processing projects are continuously entering the open source stage and are challenging the state of the art with new features and capabilities. Open source communities are constantly improving the capabilities of their projects and are pushing the technical boundaries of stream processing. We will take a brief look into the past to see where open source stream processing came from and where it is today.

A Bit of History

The first generation of distributed open source stream processors (2011) focused on event processing with millisecond latencies and provided guarantees against loss of events in the case of failures. These systems had rather low-level APIs and did not provide built-in support for accurate and consistent results of streaming applications because the results depended on the timing and order of arriving events. Moreover, even though events were not lost, they could be processed more than once. In contrast to batch processors, the first open source stream processors traded result accuracy for better latency. The observation that data processing systems (at this point in time) could either provide fast or accurate results led to the design of the so-called lambda architecture, which is depicted in Figure 1-7.

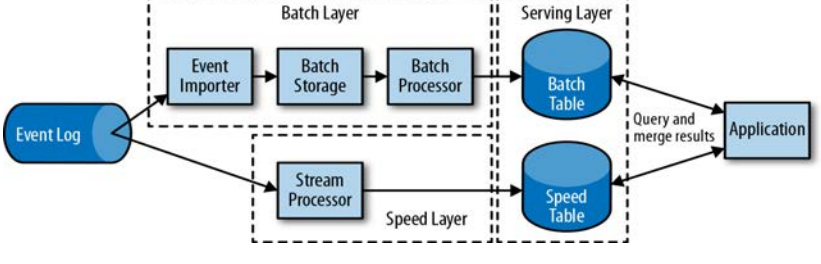

Figure 1-7. The lambda architecture

The lambda architecture augments the traditional periodic batch processing architecture with a speed layer that is powered by a low-latency stream processor. Data arriving at the lambda architecture is ingested by the stream processor and also written to

batch storage. The stream processor computes approximated results in near real time and writes them into a speed table. The batch processor periodically processes the data in batch storage, writes the exact results into a batch table, and drops the corresponding inaccurate results from the speed table. Applications consume the results by merging approximated results from the speed table and the accurate results from the batch table.

The lambda architecture is no longer state of the art, but is still used in many places. The original goals of this architecture were to improve the high result latency of the original batch analytics architecture. However, it has a few notable drawbacks. First of all, it requires two semantically equivalent implementations of the application logic for two separate processing systems with different APIs. Second, the results computed by the stream processor are only approximate. Third, the lambda architecture is hard to set up and maintain.

Improving on the first generation, the next generation of distributed open source stream processors (2013) provided better failure guarantees and ensured that in case of a failure each input record affects the result exactly once. In addition, programming APIs evolved from rather low-level operator interfaces to high-level APIs with more built-in primitives. However, some improvements such as higher throughput and better failure guarantees came at the cost of increasing processing latencies from milliseconds to seconds. Moreover, results were still dependent on timing and order of arriving events.

The third generation of distributed open source stream processors (2015) addressed the dependency of results on the timing and order of arriving events. In combination with exactly-once failure semantics, systems of this generation are the first open source stream processors capable of computing consistent and accurate results. By only computing results based on actual data, these systems are also able to process historical data in the same way as "live" data. Another improvement was the dissolution of the latency/throughput tradeoff. While previous stream processors only provide either high throughput or low latency, systems of the third generation are able to serve both ends of the spectrum. Stream processors of this generation made the lambda architecture obsolete.

In addition to the system properties discussed so far, such as failure tolerance, performance, and result accuracy, stream processors have also continuously added new operational features such as highly available setups, tight integration with resource managers, such as YARN or Kubernetes, and the ability to dynamically scale streaming applications. Other features include support to upgrade application code or migrate a job to a different cluster or a new version of the stream processor without losing the current state.

A Quick Look at Flink

Apache Flink is a third-generation distributed stream processor with a competitive feature set. It provides accurate stream processing with high throughput and low latency at scale. In particular, the following features make Flink stand out:

- Event-time and processing-time semantics. Event-time semantics provide consistent and accurate results despite out-of-order events. Processing-time semantics can be used for applications with very low latency requirements.
- Exactly-once state consistency guarantees.
- Millisecond latencies while processing millions of events per second. Flink applications can be scaled to run on thousands of cores.
- Layered APIs with varying tradeoffs for expressiveness and ease of use. This book covers the DataStream API and process functions, which provide primitives for common stream processing operations, such as windowing and asynchronous operations, and interfaces to precisely control state and time. Flink's relational APIs, SQL and the LINQ-style Table API, are not discussed in this book.
- Connectors to the most commonly used storage systems such as Apache Kafka, Apache Cassandra, Elasticsearch, JDBC, Kinesis, and (distributed) filesystems such as HDFS and S3.
- Ability to run streaming applications 24/7 with very little downtime due to its highly available setup (no single point of failure), tight integration with Kubernetes, YARN, and Apache Mesos, fast recovery from failures, and the ability to dynamically scale jobs.
- Ability to update the application code of jobs and migrate jobs to different Flink clusters without losing the state of the application.
- Detailed and customizable collection of system and application metrics to identify and react to problems ahead of time.
- Last but not least, Flink is also a full-fledged batch processor.[1]

In addition to these features, Flink is a very developer-friendly framework due to its easy-to-use APIs. The embedded execution mode starts an application and the whole Flink system in a single JVM process, which can be used to run and debug Flink jobs within an IDE. This feature comes in handy when developing and testing Flink applications.

1 Flink's batch processing API, the DataSet API, and its operators are separate from their corresponding streaming counterparts. However, the vision of the Flink community is to treat batch processing as a special case of stream processing—the processing of bounded streams. An ongoing effort of the Flink community is to evolve Flink toward a system with a truly unified batch and streaming API and runtime.

Running Your First Flink Application

In the following, we will guide you through the process of starting a local cluster and executing a streaming application to give you a first look at Flink. The application we are going to run converts and aggregates randomly generated temperature sensor readings by time. For this example, your system needs Java 8 installed. We describe the steps for a UNIX environment, but if you are running Windows, we recommend setting up a virtual machine with Linux, Cygwin (a Linux environment for Windows), or the Windows Subsystem for Linux, introduced with Windows 10. The following steps show you how to start a local Flink cluster and submit an application for execution.

1. Go to the Apache Flink webpage and download the Hadoop-free binary distribution of Apache Flink 1.7.1 for Scala 2.12.
2. Extract the archive file:

   ```
   $ tar xvfz flink-1.7.1-bin-scala_2.12.tgz
   ```

3. Start a local Flink cluster:

   ```
   $ cd flink-1.7.1
   $ ./bin/start-cluster.sh
   Starting cluster.
   Starting standalonesession daemon on host xxx.
   Starting taskexecutor daemon on host xxx.
   ```

4. Open Flink's Web UI by entering the URL **`http://localhost:8081`** in your browser. As shown in Figure 1-8, you will see some statistics about the local Flink cluster you just started. It will show that a single TaskManager (Flink's worker processes) is connected and that a single task slot (resource units provided by a TaskManager) is available.

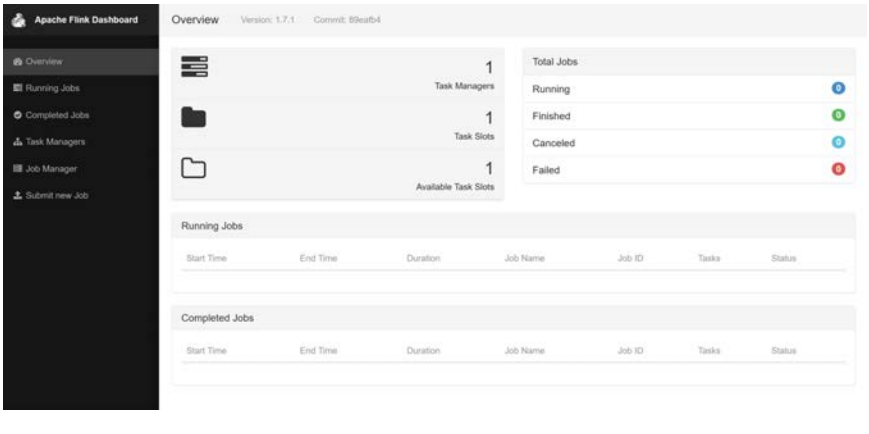

Figure 1-8. Screenshot of Apache Flink's web dashboard showing the overview

5. Download the JAR file that includes examples in this book:

```
$ wget https://streaming-with-flink.github.io/\
examples/download/examples-scala.jar
```

You can also build the JAR file yourself by following the steps in the repository's README file.

6. Run the example on your local cluster by specifying the application's entry class and JAR file:

```
$ ./bin/flink run \
  -c io.github.streamingwithflink.chapter1.AverageSensorReadings \
  examples-scala.jar
Starting execution of program
Job has been submitted with JobID cfde9dbe315ce162444c475a08cf93d9
```

7. Inspect the web dashboard. You should see a job listed under "Running Jobs." If you click on that job, you will see the dataflow and live metrics about the operators of the running job similar to the screenshot in Figure 1-9.

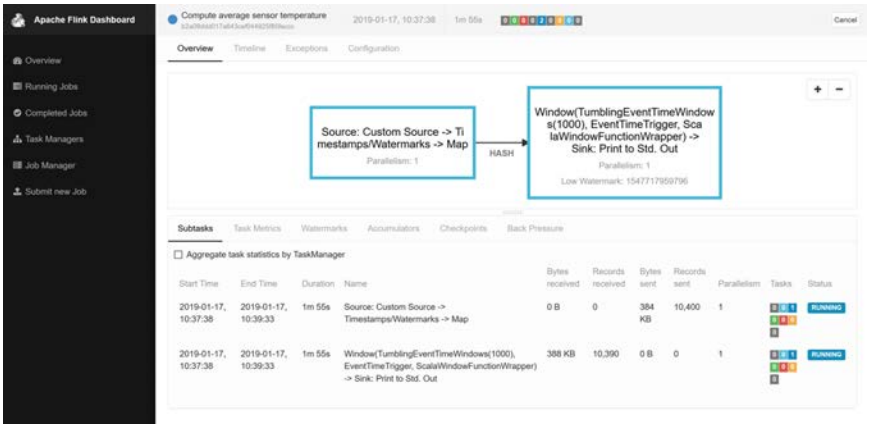

Figure 1-9. Screenshot of Apache Flink's web dashboard showing a running job

8. The output of the job is written to the standard out of Flink's worker process, which is redirected into a file in the *./log* folder by default. You can monitor the constantly produced output using the `tail` command as follows:

```
$ tail -f ./log/flink-<user>-taskexecutor-<n>-<hostname>.out
```

You should see lines like this being written to the file:

```
SensorReading(sensor_1,1547718199000,35.80018327300259)
SensorReading(sensor_6,1547718199000,15.402984393403084)
SensorReading(sensor_7,1547718199000,6.720945201171228)
SensorReading(sensor_10,1547718199000,38.101067604893444)
```

The first field of the `SensorReading` is a `sensorId`, the second field is the timestamp in milliseconds since `1970-01-01-00:00:00.000`, and the third field is an average temperature computed over 5 seconds.

9. Since you are running a streaming application, the application will continue to run until you cancel it. You can do this by selecting the job in the web dashboard and clicking the Cancel button at the top of the page.
10. Finally, you should stop the local Flink cluster:

```
$ ./bin/stop-cluster.sh
```

That's it. You just installed and started your first local Flink cluster and ran your first Flink DataStream API program! Of course, there is much more to learn about stream processing with Apache Flink and that's what this book is about.

Summary

In this chapter, we introduced stateful stream processing, discussed its use cases, and had a first look at Apache Flink. We started with a recap of traditional data infrastructures, how business applications are commonly designed, and how data is collected and analyzed in most companies today. Then we introduced the idea of stateful stream processing and explained how it addresses a wide spectrum of use cases, ranging from business applications and microservices to ETL and data analytics. We discussed how open source stream processing systems have evolved since their inception in the early 2010s and how stream processing became a viable solution for many use cases of today's businesses. Finally, we took a look at Apache Flink and the extensive features it offers and showed how to install a local Flink setup and run a first stream processing application.

O'REILLY®

Stream Processing with Apache Flink

Get started with Apache Flink, the open source framework that powers some of the world's largest stream processing applications. With this practical book, you'll explore the fundamental concepts of parallel stream processing and discover how this technology differs from traditional batch data processing.

Longtime Apache Flink committers Fabian Hueske and Vasia Kalavri show you how to implement scalable streaming applications with Flink's DataStream API and continuously run and maintain these applications in operational environments. Stream processing is ideal for many use cases, including low-latency ETL, streaming analytics, and real-time dashboards as well as fraud detection, anomaly detection, and alerting. You can process continuous data of any kind, including user interactions, financial transactions, and IoT data, as soon as you generate them.

- Learn concepts and challenges of distributed stateful stream processing
- Explore Flink's system architecture, including its event-time processing mode and fault-tolerance model
- Understand the fundamentals and building blocks of the DataStream API, including its time-based and stateful operators
- Read data from and write data to external systems with exactly-once consistency
- Deploy and configure Flink clusters
- Operate continuously running streaming applications

"Stream Processing with Apache Flink is a great book for everyone from old-timers in the streaming world to beginner software and data engineers writing their first stream processing jobs. As the book reviews Flink, it also teaches core streaming fundamentals that will help readers level up their technical thought process. Total recommended read."

—**Ted Malaska**
Director of Enterprise Architecture at Capital One

Fabian Hueske is a PMC member of the Apache Flink project and has been contributing to Flink since day one. Fabian is cofounder of data Artisans (now Ververica) and holds a PhD in computer science from TU Berlin.

Vasiliki (Vasia) Kalavri is a postdoctoral fellow in the Systems Group at ETH Zurich. Vasia is a PMC member of the Apache Flink project. An early contributor to Flink, she has worked on its graph processing library, Gelly, and on early versions of the Table API and streaming SQL.

DATA SCIENCE

MRP: ₹ 1,050 .00

Twitter: @oreillymedia
facebook.com/oreilly

SHROFF PUBLISHERS & DISTRIBUTORS PVT. LTD.

ISBN : 978-93-5213-828-9

9 789352 138289

First Edition/2019/Paperback/English