State Migration in Stream Processing Systems

Author:
Marlo Ploemen

Supervisor:
Dr. Asterios Katsifodimos

Co-supervisor:
Dr. Marios Fragkoulis

Graduation committee:
Prof.dr. Arie van Deursen

TU Delft

State Migration in Stream Processing Systems

by

Marlo Ploemen

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday October 19, 2021 at 14:00 PM.

Abstract

In recent years, the interest for serverless computing has grown tremendously. The most common form of serverless computing, Function-as-a-Service (FaaS), uses data centers of large public cloud providers to run simple functions. The cloud providers are responsible for the operational and deployment aspects. Non-trivial function implementations require state to perform the desired business logic. Current FaaS implementations using an externalized database for state cannot achieve the low latency scenarios required for some services. Previous work investigated Stateful Function-as-a-Service (SFaaS) using Stream Processing Systems as a runtime. State migration, as a result of schema evolution on SFaaS, remains an open challenge.

This thesis investigates common practices regarding schema evolution and their applicability to stream processing systems. Based on the investigation, the performed work demonstrates a schema driven approach to state migration in stream processing systems. The approach demonstrates that a view on both the source and target state schema can also yield implicit transformations for schema compatibility.

The work is demonstrated using a modified version of Apache Flink and evaluated based on common evolution scenarios and hypothesized changes to real world queries from the NEXmark benchmark.

Student number:	4276906	
Project duration:	Sept 1, 2020 – Oct 19, 2021	
Thesis committee:	Dr. A. Katsifodimos,	TU Delft, supervisor
	Dr. M. Fragkoulis,	TU Delft, co-supervisor
	Prof.dr. A. van Deursen	TU Delft, graduation committee

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Preface

I would like to thank my supervisor, Dr. Asterios Katsifodimos, and co-supervisor, Dr. Marios Fragkoulis, for their guidance during the thesis. I am grateful for the many discussions we had on the topic and the questions that led to critical re-evaluations. I would also like to extend my gratitude to all members of the AI4Fintech research group at ING. The feedback and suggestions offered during the weekly group meetings were extremely useful. Finally I want to thank my friends and family who supported me during my study.

Marlo Ploemen
Delft, October 2021

Contents

1 Introduction

In the 1940s, first generation software was written using binary instructions presented as punch cards [93]. Even though the costs of running the software was very high, the need for the computing power outweighed these costs. Years of research and development led to increasingly higher densities of components on chips (with the rate of change often dubbed as "Moore's law" [76]). Programming languages and compilers hide the complexities of translating the requirements of software to the actual executed machine code. Some high-level programming languages trade the ease of writing with the efficiency of the executed code, and some low-level programming languages vice-versa. The load on software services often requires a higher compute capacity than can be supplied by a single machine. In line with the emerging cloud architectures, services are hosted in data centers of public cloud providers. The cloud architecture model allows for a high degree of flexibility in compute power by adding or removing servers, with certain cloud providers even scaling automatically. Running software services on multiple machines introduces distributed computing complexities, amongst other challenges. This led to the development of 'serverless' computing. In serverless computing, the operational and deployment challenges are handled by a third-party.

The Function-as-a-Service (FaaS) model is an implementation of the 'serverless' model which allows developers to only describe the business logic of a software service. This enables developers to quickly grow a software service to planet-scale using the digital infrastructure of public cloud providers without managing any of the deployment characteristics. In a way, the complexities of current software deployment models are abstracted from the developer who is, like in the 1940s, tasked with writing business logic.

With parallel compute power no longer being a limitation for software services, data driven software services encounter other challenges. Three notable challenges are maintaining a consistent state of the software service, state locality and coordinating inter-service function invocations [53]. Most non-trivial computations require some form of state. For example, calculating a rolling average where the state represents the current average which is updated on function invocations. Such calculations seem trivial but require careful coordination of updating the state in distributed systems. When the state is stored in an external system, every function invocation has to read the state, perform some calculation and then write back the result [3]. The latency introduced by the absence of state locality poses challenges for time constrained services.

It can be concluded that there is a need for Stateful-Function-as-a-Service (SFaaS) implementations which solve the complexities of dealing with (distributed) state. Stream processing systems faced challenges of parallelizing workloads and pose as a good candidate for an SFaaS runtime [53].

In section 1.1, the role of cloud providers in current day-to-day deployments is described. section 1.2 describes the need for a 'serverless' model and the challenges it faces. In section 1.3, open challenges for using stream processing systems as a runtime for SFaaS are detailed. Finally the chapter concludes with the problem statement, approach, research questions and an outline for the remainder of the thesis.

1.1. Cloud Providers

Using the data centers of public cloud providers, organizations can quickly provide software services to large audiences at various locations around the globe. In a self managed scenario, this would require high upfront cost and specialized teams to maintain the compute infrastructure. In contrast, cloud providers use a pay-as-you-go model where only the usage of the machines is billed with no upfront commitment. Data centers of public cloud providers are strategically laid at various geographical locations such that the client-to-server latency is minimal and to allow the tenants to comply with local legislation.

The underutilization of physical machines led to virtualization [43]. Virtualization allows cloud providers to share the same physical machine with multiple tenants which are isolated from each other. Organizations can create an image containing all components needed to run the software service. The cloud providers use the supplied image to create the virtual machines. Due to the rapid growth of data centers worldwide the costs of using a cloud provider to run and maintain virtual machines has decreased [50].

A survey conducted by NGINX found that 86% of the interviewed I.T. professionals use a public cloud provider for their digital infrastructure [63]. Data centers of public cloud providers allow corporations to grow to a planet scale, offering e.g. Netflix – an internet television network – to use over 100.000 server instances to operate in more than 190 countries with over 200 million members [60].

Containerization, lightweight virtualization, reduces the overhead of virtual machines [43]. Container images contain the static runtime environment in which a software service is executed. Docker models the steps to create an image as stacked layers. Containers share the host operating system and kernel so the first layer (the 'base' layer) should be identical. An advantage of the layer model is that generic runtime environments can be created which are specialized in multiple different environments. For example a container image with Python 3.6 and its dependencies preinstalled where a developer only has to supply the source code. The generic images can be shared on public repositories such as DockerHub [1].

A Dockerfile describes the creation of Docker container images. Listing 1.1 illustrates how the runtime environment can be described in only 6 lines.

Listing 1.1: Example Flask Dockerfile [22]

```
FROM python:3.8-slim-buster

WORKDIR /app

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY . .

CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

Containers thus (as shown in Listing 1.1) allow fine grained control over a generic runtime environment. Containers however do not not solve the maintenance and operational costs of the digital infrastructure. In some situations the source code that performs the business logic is trivial compared to the operational challenge of setting up the digital infrastructure running the application [50].

1.2. Serverless Computing

Container images allow flexible creation of runtime environments. Combined with a digital infrastructure to rapidly deploy these images, allows for a higher flexibility of architecting applications.

[1] https://hub.docker.com/

Managing a large scale digital infrastructure, however, is non trivial. In recent years, there has been an increasing interest in serverless computing, described as the next step in cloud computing [18] with many public cloud providers offering serverless capabilities. In serverless computing, the operational aspects of scaling an application are handled by a third party [66]. Serverless architectures are of particular interest in scenarios where the business logic of a data driven application can be written in a few dozen lines of code which is a fraction of the effort to deploy the code [50].

The most common form of serverless computing is Function-as-a-Service (FaaS) [3]. In a FaaS deployment architecture, only the business logic of function is supplied by a tenant. The management of the runtime and deployment of the functions is handled by the serverless platform provider [66]. A common misconception about serverless computing is that no servers are needed but this is not the case [66]. The management and operational challenges are simply abstracted and invisible to the tenant.

Current FaaS offerings are stateless. This relaxes the deployment characteristics because a system external to the FaaS scope handles the state of the application. In order to resemble state, the function can read some data D from storage, process D and write back to external storage [3]. In data-driven serverless applications, state can be stored in a central database management system [25] where data has a well defined schema.

Due to the stateless nature of FaaS, horizontal upscaling and horizontal downscaling can be achieved by adding or removing nodes from a load balancer. Compute capacity is allocated ad-hoc to incoming requests. The pay-as-you-go model applies to incoming requests and the time it takes to handle the request instead of raw provisioned resources such as a CPU and memory. Because the operational aspects are handled, developers can view their application as orchestrating a series of function calls instead of provisioning servers.

Challenges of Serverless Computing

In order to deliver on the serverless computing premise, serverless platforms need to handle all deployment characteristics. This leads to challenges for both the tenant and the provider. A serverless platform might lack an implementation detail [18] such as a programming language or library. With little control over the deployment characteristics, the tenant is dependent on the provider for up-to-date dependencies on the platform. Packaging serverless function as containers, e.g. using Docker [43] [18], can provide the tenant with control over the dependencies with a provider defined public API.

A tenant needs to understand how the application is being deployed to understand the constraints of the application. Providers have various APIs to provide control over the process, as these processes are non-standardized this can lead to vendor lock-in [18].

State is stored external to the function [3] [18]. The function reads from an external database management system, processes the data and writes back to the database management system. This introduces additional latency and is transforming the application from compute intensive to I/O intensive [2]. Furthermore, coordination is hindered by the lack of a managed state [3]. In Figure 1.1, a visualization of a FaaS model is shown. The figure illustrates that replicas of the same function are scaled within the same bounded context (not local) and functions have access to a datastore specific to the function. This is not a requirement but purely for illustrative purposes. The coordination of function invocations, e.g. a function processing an ordered set of events, can only be guaranteed by implementing an orchestrator responsible for orchestrating function invocations.

Amazon Web Services (AWS) offers the AWS Lambda serverless compute service [16]. AWS is a FaaS implementation where a ‘Lambda’ refers to a function. Lambda functions can be orchestrated using AWS Step Functions [17]. The orchestrator models events passing through a set of lambdas as a state machine. State in the AWS Step Function context refers to the state in a state machine, not a view on past events. Figure 1.2 illustrates this using four functions which are conditionally connected.

Figure 1.1: Example FaaS layout

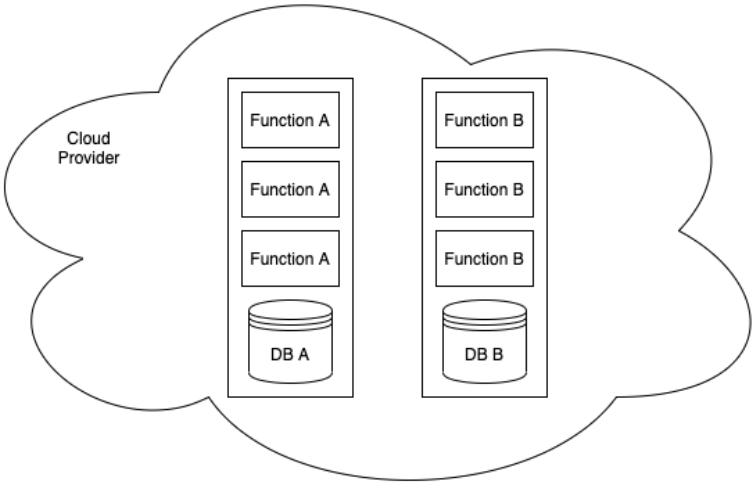
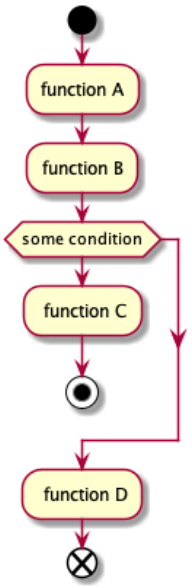


Figure 1.2: FaaS workflow as state machine



1.3. Stream Processing Systems

Inter-function interactions modeled as a workflow and the lack of local state resemble issues solved in the stream processing domain. Akhter, Fragkoulis, and Katsifodimos [3] show that a modified version of a stream processing system can be used as a runtime for *stateful functions*. The state machine model for FaaS function, visualized in Figure 1.2, can be translated to the dataflow graph model used in stream processing systems.

Stream processing systems have been widely used for analytical tasks. Implementations of stream processing systems can be found in numerous sectors where the enormous amount of data being generated cannot be stored for later processing [28]. Event processing systems can be found in many industries including health informatics, astronomy, telecommunication, electric grids, energy, geography, transportation [28], financial industries such as algorithmic trading [73] and fraud detection [58]. Stream processors are implementations of event driven architectures (EDAs). Event driven architectures in general support a fault tolerant state with support for event partitioning and scaling out [53]. These properties make stream processing systems a candidate as a backend for cloud applications and microservices [53] [24].

Stream processing systems such as Apache Spark employ micro-batching to simulate streaming pipelines. Micro-batching increases the latency since events are buffered but also increases throughput [88]. Batch processors have been widely used alongside stream processors in what is referred to as the Lambda Architecture [54]. A Lambda Architecture comprises of three layers: a speed layer for latency sensitive processing, a batch layer, and a serving layer to be queried.

Due to the support for event partitioning and scaling out, stream processors are often distributed. The pipeline running on a stream processor can be represented as a directed acyclic graph (DAG). Business logic is processed at each node of the DAG while edges represent the passing of data. The physical implementation of the DAG can contain multiple parallel operators running the business logic at each node. For consistency of the results, the stream can be partitioned based on attributes of the events. Such a model allows for fine grained control over the parallelization of the work.

Challenges of Stream Processing Systems as a runtime for SFaaS

Leveraging stream processing systems as a runtime for SFaaS imposes additional advanced requirements [53]:

- **Coordination —** of transactions such that the state remains consistent.
- **Local State —** to avoid the additional latency of network calls to read and write state.
- **Global State View & Analytics —** for insights on the stored data.
- **Loose Coupling —** to retain the advantages of micro-services.
- **Debugging & Auditing —** to verify the integrity of the service.
- **Dynamic (Re)configuration —** to perform updates on the (running) service.

Software tends to be a rapidly changing environment. In 2014, Amazon's Apollo Deployment engine was responsible for 50 million deployments in 12 months [89]. A more recent study [75] indicates that similar practices are used by other big tech companies such as Facebook, Google, Netflix, and OANDA.

The absence of stateful reconfiguration is reiterated by Heus, Fragkoulis, and Katsifodimos [44] where *migrate tasks* for *stateful* functions are proposed as an open challenge.

1.4. Problem Statement

Stateful reconfiguration is one of the open challenges for stream processing systems to be used as a runtime for SFaaS. Reconfiguration can be the result of hardware changes or changes to the service itself [53]. The reprocessing of state during schema evolutions is often executed by replicating the same code which creates the streaming job [26].

To support reconfiguration of stateful dataflow graphs, the main research goal is describing a process which reflects on the previous runtime configuration. This process should validate the reconfiguration and orchestrate the transition into the new configuration.

The common practice of schema definitions in relational database models will be used as an intermediate representation. The following topics are of interest to aid the research goal:

- **P1: Schema Definition —** the attributes and shape of a schema required for the reconfiguration of a service.
- **P2: Schema Transformation —** where an instance of a schema is mutated to reflect a change between a source and target version.
- **P3: Schema Evolution —** of identifying changes between two instances of a schema. These changes can be checked to validate the schema compatibility.
- **P4: Data Migration —** how data is migrated from the source state to the target state in order to comply with an evolved schema.
- **P5: Deployment Semantics —** how the reconfiguration impacts deployment semantics.
- **P6: Processing Semantics —** how the reconfiguration impacts the processing semantics and data integrity.

1.5. Approach

To achieve this goal, a modified version of Apache Flink is presented [2]. Apache Flink is a stream processing system providing mechanisms for guaranteeing exactly once processing in a distributed system.

As a starting point, version 1.12 of Apache Flink will be used. The used version contains mechanisms to incrementally snapshot state and generate savepoints [25], a library for manipulating savepoints, and mechanisms for stop-and-restart deployments based on a savepoint. The used version does not feature schema generation, nor a mechanism for unifying schema evolution, state migration, or job deployment. The main contributions of this thesis are:

- **schema generation —** a mechanism for deriving schemas which represent a stateful dataflow graph
- **schema comparison —** a comparison of two instances of a derived dataflow graph schema
- **schema transformation —** a mechanism for incrementally transforming a dataflow graph schema based on supplied migrations

The implications of the contributions are shown with a REST endpoint which controls the stateful dataflow graph schema evolution and state migration based on a source and target version. Deriving implicit migrations where possible to bridge incompatible dataflow graph schemas based on schema comparison, and a declarative API to explicitly specify changes to the stateful dataflow graph schema.

1.6. Research Questions

With (re)configuration at the heart of the problem statement and a view on the advanced SFaaS requirements, this thesis will investigate a code-to-schema based approach to reconfiguration. This combines the stable expressiveness of the relational database model with the processing power of distributed local state event streaming platforms. To achieve this, the following four research questions are defined:

- **RQ1 —** How can a stateful dataflow graph be transformed to a state schema?

[2] https://github.com/delftdata/msc-state-migration-streams

- **RQ2 —** Given a state schema for a stateful dataflow graph, what evolution scenarios exist?
- **RQ3 —** Which transformations can be derived without user intervention that migrate state for stateful dataflow schema evolutions?
- **RQ4 —** How can two instances of stateful dataflow graph schemas be used for (dynamic) reconfiguration?

1.7. Outline

In chapter 2, a background and/or related work on migration constraints, continuous deployment, deployment strategies, schema definition, schema evolution, state migration, functions as a service, and stream processing systems is discussed. The chapter 2 will act as a review on the current state of the research and best practices used in the industry. In chapter 3, the performed work is elaborated. First by describing the general outline of the work into separate components, after which each sub-component is discussed separately.

In chapter 4, the evaluation metrics, experiments and results are described before concluding the research in chapter 5. Finally, chapter 6 contains the contributions of this thesis and pointers for future work.

2

Background and Related Work

In this chapter, related work and background information on various topics related to the research is presented. In section 2.1 a background is provided on migration constraints. Migration constraints impose non-functional requirements on applications and their evolutionary lifecycle. In section 2.2, a general outline of continuous deployment and its use-cases is described. Various deployment techniques are described in section 2.3. In section 2.4, the definition and evolution schemas are described. Building on the schema definition, section 2.5 describes the underlying physical state migration. Finally the chapter concludes with a background on functions as a service (section 2.6) and stream processing systems (section 2.7).

2.1. Migration Constraints

Due to business requirements, certain applications can be required to be highly-available. Highly-available configurations require that all parts of the application are redundantly available such that during failure of one part, another part can take over. The degree of high-availability is the degree of failures that an application can sustain. The corresponding migration constraint is defined as:

- **MC1: Availability**

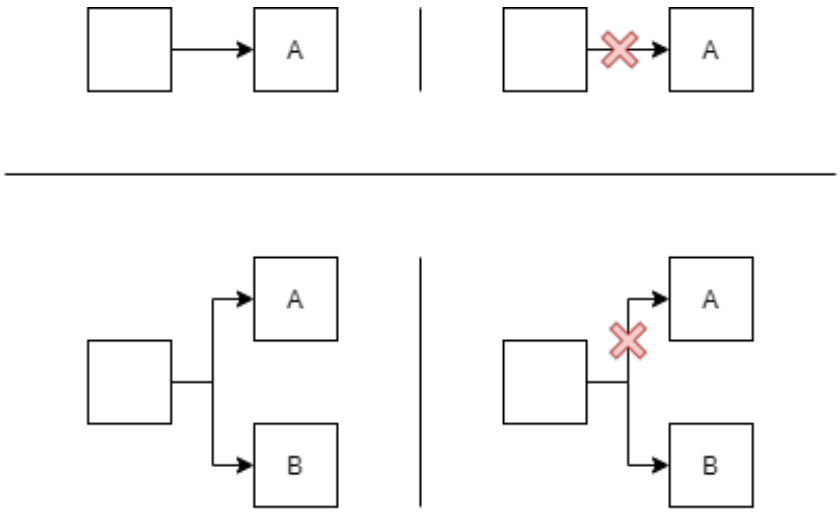
To support availability requirements, load balancing can be introduced. Load balancers distribute the load of an application to multiple compute units that perform the business logic. By performing period health checks on the compute units, whenever there is a failure the load balancer will not send additional requests to the failed compute unit. Alternatively, a system not able to distribute load can minimize downtime using hot-standby components. When the primary system fails, another system takes over. In case of a database, there could be a read-replica. All writes are performed on the primary database but read operations can occur from both the primary database and the read replica. Whenever there is a failure in the primary database, the read-replica can be promoted to become the new primary database.

The constraint can be measured in the degrees of failure that can be sustained. In Figure 2.1 a 0-degree and 1-degree system are visualized. The degree of failure of the component balancing the load is excluded from the figure. The top figure illustrates that when A cannot be used to process requests (the node failed, is updating, the traffic cannot reach the node, etc.), there is a service outage. The bottom illustrates that introducing a secondary component mitigates this 0-degree failure as requests can still be routed towards B.

Table 2.1: Compatibility Types [77]

Compatibility type	Allowed schema changes	Previous versions	Upgrade first
Backward	Delete fields, Add optional fields	last	consumers
Forward	Add fields, Delete optional fields	last	producers
Full	Add optional fields, Delete optional fields	last	any order
Backward transitive	Delete fields, Add optional fields	all	consumers
Forward transitive	Add fields, Delete optional fields	all	producers
Full transitive	Add optional fields, Delete optional fields	all	any order
None	All	n/a	n/a

Figure 2.1: 0-degree (top) and 1-degree (bottom) high availability setup



When a service exposes a public API, there might be consumers depending on the service. This relationship can be described as a producer-consumer relationship. Additional migration constraints are imposed on a service in a producer-consumer relationship. The evolution of the producer must be supported by the consumer based on the defined compatibility type. Seven compatibility types of the producer-consumer relationship can be defined depending on the type of change [77]. The corresponding migration constraint is defined as:

- **MC2: Schema Compatibility**

The described compatibility types in Table 2.1, relate to a source and a target version of the schema, the producer and the consumer. Respectively S_s, S_t, P_s, P_t, C_s and C_t. With backwards compatibility, the implication is that a consumer C_t using schema S_t should be able to interpret events produced by P_s with schema S_s. When $S_s \rightarrow S_t$ deletes a field, C_t can ignore the dropped field and continue processing. Adding optional fields during $S_s \rightarrow S_t$ can be initialized as a missing value in C_t as the schema states it to be optional. With the change $S_s \rightarrow S_t$ categorized as backwards compatible, the consumer can safely update. When the compatibility requirement is transitive, it should hold for all previous versions. The inverse relation between producers-consumers is described by forward compatibility where fields can be added and optional fields deleted. A hybrid compatibility model being both forward and backward compatible is referred to as *full* compatible.

2.2. Continuous Deployment

Continuous deployment refers to the process of automating software artifact deployments. Often in large software projects there are multiple environments where the software is deployed. These environments correspond to different stages of testing and compliance of releasing new software artifacts. For example a development environment to identify integration errors, an acceptance

Figure 2.2: Simple continuous deployment pipeline

Table 2.2: Deployment challenges

Availability	Local State	Challenge
no	no	n/a
yes	no	must ensure multiple running instances
no	yes	must migrate local state
yes	yes	consistent local state, processing semantics

environment for user acceptance testing, or a production environment where no bugs should exist at all. Manual deployment to these various environments can be a time consuming, complex process which can introduce blocking factors such as the availability of the developers responsible for the deployment.

Continuous Deployment implicitly documents this process as code and allows for more frequent deployments. Benefits include the deployment without relying on key developers, saving time and having a complex pipeline documented. Additionally, when a release introduces bugs (either regressions or new bugs), it is easier to trace these bugs to the newly deployed code due to the lower amount of changes versus large software artifact releases. The manual processes which act as barriers for the transformation towards agile development can be seen as technical debt [85]. There are different stages to agile development as stated by Olsson, Alahyari, and Bosch [64]:

1. Adopting agile practices in R&D
2. Continuous integration including automated tests and builds
3. Continuous deployment of the software artifact to a customer environment
4. Responding to instant feedback from customers

Typically, continuous integration is performed when a developer commits changes to a version control system. The version control system is capable of statically analyzing the source code, creating builds and running tests. When all actions succeed, the system can deploy the software artifact to a target environment [51]. A simple pipeline is visualized in Figure 2.2. If one of the steps in such a pipeline fails, the developer introducing the change to the system can be notified of the failure. With a version control system at the heart of the continuous deployment pipeline, the requirement of accessible software artifacts [47] can be guaranteed.

2.3. Deployment Strategies

During the deployment step defined in section 2.2, the service is deployed. Based on the migration constraints defined in section 2.1, various mechanisms for deployment exist. Additionally, resource usage, local state, and processing semantics also influence the used deployment strategy. Resource usage refers to the compute capacity that can be allocated during the deployment, local state refers to the non-externalized state, and processing requirements refers to the processing of the underlying service, a requirement further elaborated upon in subsection 2.7.4.

Compute capacity limits the available deployment strategies while requirements on local state and availability are complex to combine. In Table 2.2, the combination of the availability and local state requirement describes the challenges the deployment faces.

A highly available stateless deployment is visualized in Figure 2.3. Figure 2.3 also illustrates the concept of resource usage and MC2, the compatibility requirement. The deployment strategy can be referred to as rolling blue/green deployment strategy. The resource capacity is two with no

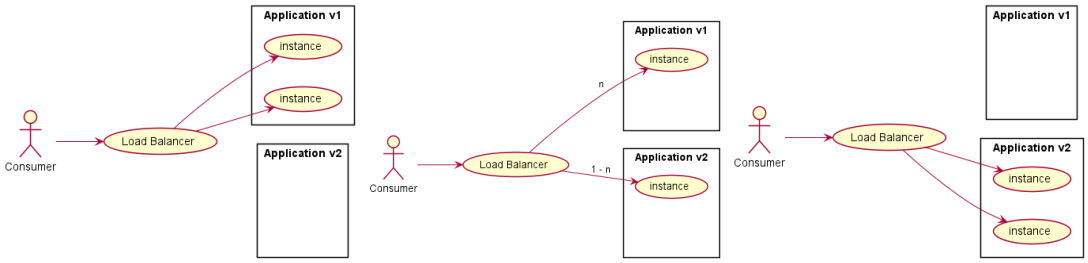


Figure 2.3: Deploying a new version of a stateless application

Table 2.3: Deployment strategies

Deployment Strategy	Staged
In-place [30]	no
Canary [32] [30]	no
Blue-green [75] [36] [30]	yes
Dark launches [75]	yes
Staging/baking [75]	yes
All-at-once [30]	no
Linear Deployment [30]	yes

temporary over provisioning allowed. Based on these constraints, an instance of the initial (source) version is stopped. A new instance (target) version can now be started with the newly available resources. The load balancer assuring high compatibility can now balance traffic to both running instances. This imposes the MC2 requirements to be handled. Finally, the last source instance is stopped and restarted as the target version.

Deployment strategies can be categorized as depicted in Table 2.3. In-place migrations replace the service on the already allocated compute instance. In a cluster setup this can be performed while guaranteeing high availability using a rolling deployment [30]. Canary deployments are a risk averse strategy to deploy new versions [32]. A fraction of the running instance is updated to the target version and monitored for faults. In the event of a fault, the deployment is cancelled. In a blue/green migration, two almost identical environments are used [36]. Based on a load balancing policy, traffic is shifted to blue or green versions of the application of which one is ahead. In the event of failure, the traffic can be shifted to the prior version.

2.4. Schema Definition and Evolution

In data-driven serverless applications, state is often stored in a central database management system [25]. To achieve low latency scenarios, distributed streaming systems often store state local to the operator processing the state. State is therefore inherently sharded as well. Injecting special control messages into the event stream can produce a consistent materialized global view of the state [25].

A recent survey on Stack Overflow shows that the four most popular database technologies, both among all respondents and when filtered on only professionals, are relational databases [81]. Relational databases use a tabular structure where tables represent some entity with columns as the attributes. Rows are than structured datapoints of the entity.

Relational databases use SQL to retrieve results or manipulate the schema [71], often with some technology specific dialect. Data Definition Language (DDL) is a subset of SQL statements which refers to manipulating the structure of the database. These statements include (with a relevant example):

- **CREATE —** to create tables with a defined set of columns
- **ALTER —** to alter existing table definitions

- **DROP —** to drop existing tables

With these three statements, it is possible to describe the schema of a database as a set of tables and their corresponding column definitions. Future schema changes which are not additions can be handled by running ALTER or DROP statements on the existing schema.

2.4.1. Using Relational Data in Applications

While the tabular-relational structure aids the data definition and data manipulation, there is an impedance mismatch between applications written in an object-oriented style that have to map to a relational data structure. This mismatch is also called the Object-relational impedance mismatch [49]. In order to store objects in a relational database management system, the persistent attributes of an object have to be mapped to columns in a table.

Object Relational Mapping (ORM) bridges the gap between the two runtime contexts of persistence and application code. Data is often persisted in a relational form where ORM abstracts the implementation of mapping application code objects to relational data, and the communication with the persistence layer of an application. Implementations of ORM frameworks can be found in many common programming languages, for example, but not limited to, Java, C++, Python, C#, PHP, JavaScript, and Ruby [86].

To facilitate bridging the impedance mismatch, ORMs often facilitate DDL functionality to create the initial schema. Evolution of the schema is often left as an exercise for the developers [21]. When creating a schema to persist data, the goal is to reflect the target environment as accurately as possible. While the assumption is that the modeled schema is stable even under a changing environment, this is often not the case [70]. To maintain compatibility between the different versions of a service and the database, schema evolutions should be tied to the application. Tools such as Liquibase [41] and Flyway [39] provide mechanisms to run a series of schema evolutions based on the current state of the database. QuantumDB [51] uses a non-blocking approach for continuous deployment pipelines using the concept of changesets.

Three techniques to schema evolution in database environments are identified by Michel, Andany, and Palisser [59]. The database schema is evolved without keeping the previous schema (ignoring consequences on data), or the schema is evolved and the data transformed into the new schema, or both versions of the schema are kept (either with historical or with parallel access). ORION is a prototype object oriented database system that was developed with schema evolution in mind [19]. ORION defines a formal framework for schema evolution and invariants which are required to hold.

A methodology is proposed by Shneiderman and Thomas [78] to automatically migrate a relational model using relational algebra. Similarly, McBrien and Poulovassilis [56] detail a methodology using hypergraph data models. If upward compatibility is required, migrating the data is inadequate as older clients cannot use the newer version of the data [57]. Program Independency [70] relates to the property that old schemas should still be queryable while the data is kept up-to-date.

2.5. State Migrations

When a state schema, a subset of the general schema related to state, has evolved, the physical state should be migrated to match the new schema. When physical data changes in database management systems, often a locking mechanism is introduced [29] to support concurrent operations not related to the current update. In section 2.4, the usage of SQL is outlined through which relational database queries are executed. To manipulate or migrate state in relational databases, DDL (discussed in section 2.4) and Data Manipulation Language (DML), a subset of statements from SQL, are used. Three key statements for DML are:

- **INSERT —** to insert new rows in the database
- **UPDATE —** to update existing rows in the database
- **DELETE —** to delete existing rows from the database

Table 2.4: Altering column type in PostgreSQL [68]

Implicit	Explicit
ALTER TABLE <some table name> ALTER COLUMN <some column name> <datatype>;	ALTER TABLE <some table name> ALTER COLUMN <some column name> <datatype> USING <expression>;

Combined with the Data Query Language (DQL) statements, these statements can ensure that the data stored in a relational database is complacent with an evolving schema. Backwards incompatible schema evolutions can be handled using an expand-contract type of query [51].

An example query syntax is provided in Table 2.4 for both implicit and explicit conversions using the PostgreSQL SQl dialect.

Because of the coupling between data driven applications and their persistence layer, changes to schemas are impacted by compatibility requirements. In Object-oriented databases, the problem of schema evolution manifests itself as type evolution [79] [19].

2.5.1. State migration in distributed dataflow graphs

The already defined evolutionary nature of services and application also translates itself to event stream processors. The challenge in event streaming engines is that the services often continuously process data that is processed in a distributed model. During state migration, the system cannot produce inconsistent results [92]. The challenge of state migration can be described as a *how* (how to migrate state) and *what* (what to migrate) [84] [31]. Often however, the underlying problem is minimizing migration *cost*:

Planning: relates to the phase before the execution of a state migration. During planning, the system analyzes the source and target state in order to describe a migration plan. Ottenwälder et al. [65] describe migration based on pre-planning in order to orchestrate migration in a complex event processing system. The *Migration Plan* model describes the time and location at which an operator should be migrated, and the required resources.

(Dynamic) reconfiguration: when processing continuous streams of data, (state) migration can be the result of a change of the DAG describing the dataflow graph. In streaming systems where the state is distributed, there are three modes of state migration [45]: stop-and-restart, partial pause-and-resume, and dataflow replication. Both stop-and-restart and pause-and-resume type of migrations pause the current processing of the event stream to various degrees. Dataflow replication can remedy this situation but increases the complexity of guaranteeing processing semantics. Floratou et al. [35] describe the reconfiguration process to self-regulating, self-stabilizing, and self-tuning dataflow graph through policies. A health manager detects symptoms based on collected metrics which are diagnosed and turned into resolutions. Where the DAG would continuously process events, special control events can be injected into the event stream to control operators [55] [20] similar to how consistent checkpoints can be generated [25]. Chi [55] describes control events to be used for continuous monitoring, dataflow reconfiguration, and auto parameter tuning of operators. Based on the collected metrics, an upstream operator could reconfigure the partitioning of the key space of downstream operators based on the imposed load. Bartnik et al. [20] use a similar mechanism but investigate the usage on introducing new operators, migrating operators, and changing operator internals. Similarly, Zhu, Rundensteiner, and Heineman [92] propose two strategies to ease migrations while guaranteeing high availability. The first *moving state* strategy relates to moving identical state between old and new versions of operators while the second *parallel tracking* strategy interconnects events such that during migration the system can continuously process events. Ding et al. [31] uses the concept of an *Migration Manager* (MM), *retriever* and *rerouter*. The migration is performed progressively with each sub-migration using a *wait* buffer during migration. This is an example of the *pause-and-resume* type of migration. Misrouted tuples are rerouted through the rerouter for correct processing. Finally Feng, Huang, and Wu [33] provide a *key-and-state* and *membership* replication scheme using a *Multilevel Counting Bloom Filter* (MLCBF) data structure to decrease the cost of stateful replication.

Cost: can be expressed as the cost of the migration. Ding et al. [31] focus on deriving the optimal placement of new operators to minimize the *migration cost*. Pietzuch et al. [67] describe the collected metrics as *cost space*. Based on the cost space, reconfiguration of the operators can be performed when the cost savings of such a migration are higher than the *minimum migration threshold* (MTT).

2.5.2. Developer interaction with streaming managed state

During the migration from a source streaming job to a target streaming job, the underlying state should be migrated. The discussed relational database domain handles these migrations using declarative SQL with occasional USING for complex migrations. Six popular streaming services are discussed related to their exposure of managed state:

- **Apache Flink:** can manage state in in-memory, HDFS, or use RocksDB. Using a barrier based checkpointing mechanism, RocksDB state can be persisted to HDFS. A similar mechanism can be used to generate savepoints [25]. Savepoints can be modified using the Flink State Processing API [82]. New operator state can be added to an existing savepoint (for new operator unique identifiers) or a savepoint can be created from scratch. To use the state processor API, developers construct a DAG similar to how a normal streaming job is built.

- **Apache Storm:** uses an external state model based on Redis [48]. As Redis is a *key-value* store, the Apache Storm only natively features *key-value* state.

- **Apache Spark:** uses in-memory state which is persisted for fault tolerance to HDFS [80]. Community created state serialization tooling exists such using a RocksDB mechanism [42] like Apache Flink.

- **Apache Heron:** compatible with Apache Storm [15]. State can be persisted to local file system or managed by Zookeeper.

- **Apache Samza:** stores state co-located to the operator using the state [74]. Provides fault tolerance by replicating the state to persistent storage. The natively supplied storage mechanism builds on LevelDB, on which RocksDB is built [38].

- **Apache Kafka Streams:** stores state on a local database using RocksDB. Kafka Streams does create a changelog topic on Kafka which is used for fault-tolerance [52].

Chen et al. [26] categorizes the mechanisms used by Flink and Spark to reprocess state as *same code*, Storm and Heron as *same DSL*, and Samza as *no batch*.

2.6. Functions-as-a-Service

One of the recent focus domains for stream processing is as a backend for scalable cloud applications [24]. Managed offerings of cloud providers deliver on the operational aspects of stateless applications but provide limited to no offerings for stateful applications [3].

A simple model of a FaaS application is that of a single function. The function is independently able to perform an entire use case. For any non trivial application, this model is either too simplistic or the application evolves into a large monolithic application. In recent years, the trend has been to steer away from large monolithic applications. This realizes the following advantages [61]:

- **Parallel development —** as developers only need to understand a small part of the codebase.

- **Independent Non-Functional Requirements —** as the used technologies, deployment characteristics and programming language can be decided on a per-service basis.

- **Scaling —** as the services can be scaled independently to account for the load.

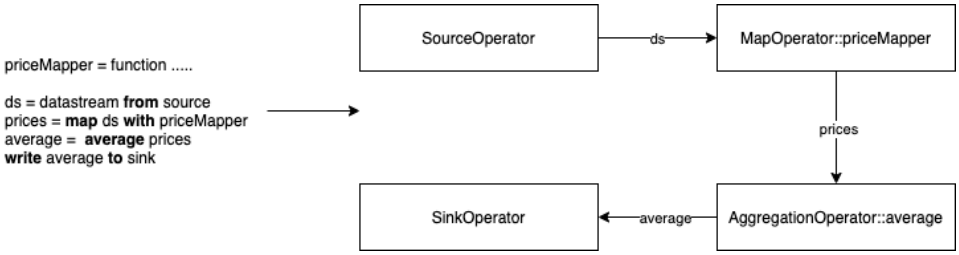


Figure 2.4: Pseudocode to Dataflow Graph

When the codebase is sharded over individual functions each providing a unique set of business logic, these functions can be referred to as micro services. A single function in a FaaS architecture is an instance of such a micro service. For the non-trivial application, many of these micro services tend to be deployed. As not a single micro service is able to deliver on the entire business case of the application, there needs to be cooperation and coordination of these functions. Amazon Step Functions, Azure Durable Functions, and IBM Composer [37] are additional services offered to provide support for coordination.

2.7. Stream Processing Systems

Interactions with systems can be modeled as events. An ordering of such events can be referred to as an event stream. A system where interactions are modeled as events implements an Event Driven Architecture (EDA). EDAs should have a fault tolerant state and support horizontal scaling [53]. Stream processing systems are implementations of such an EDA. Event processing systems are able to handle trillions of events, terabytes of state, and run on thousands of cores [14]. Traditionally, events were processed in batches on a schedule. As batches contain a finite amount of events, these can be referred to as bounded streams. A bounded event stream can trivially compute aggregations by ingesting all data and assuming that the batch is only processed after all events that are part of the batch have arrived, does not suffer from out-of-order processing. Lately, in contrast with batch processing, events can be processed in realtime. These streams can continuously receive events, without an explicit event that closes the stream, these streams can be referred to as unbounded. For the remainder of this chapter, the realtime aspect, i.e. unbounded streams, will be discussed.

2.7.1. Overview of Stream Processors

On a high level, a stream processing system functions by representing the business logic applied during the lifetime of an event as a dataflow graph. A dataflow graph is a directed graph where nodes without incoming edges or without outgoing edges are respectively referred to as sources or sinks. Nodes represent operators, performing business logic on the incoming data while the edges represent the passing of events from one node to the next. Figure 2.4 provides an example translation from pseudocode to a dataflow graph.

In a logical dataflow graph, only the computational logic is represented as part of the graph. As stream processors are often distributed systems, a physical dataflow graph represents the physical execution of a logical dataflow graph on the system. The stream processor handles the conversion from a logical dataflow graph based on supplied data parallelism and compute parallelism strategies [46] to a physical version, to be executed on a distributed system.

2.7.2. Operations in dataflow graphs

Operators in a dataflow graph can be either stateful or stateless. Like other domains as the current FaaS domain, stateless provides advantages with respect to scaling and fault tolerance. Additional compute can simply be allocated and failing operators restarted. Stateful operators however maintain internal state of past events to perform their business logic. There are four categories of operators [46]:

- operators which are responsible for communication with external systems, e.g. sources and

sinks. These nodes are respectively defined in the dataflow graph as nodes without incoming edges or without outgoing edges.

- operators which transform the data stream. Transformations use the incoming edge(s), apply some process function on the events, and produce outgoing edge(s). Transformation operators can be used to describe fan-out or fan-in patterns in dataflow graphs.
- operators which compute rolling aggregations on the data stream. The rolling operation uses the state of historic events that have been processed along with incoming events to compute a new aggregated value.
- operators that window the data stream. The data stream by default is unbounded. Windowing operators use markers in the data stream to provide bounded windows based on a windowing strategy.

2.7.3. State Management

State refers to values stored on the operator that resemble a view on past events. Stateful operators can use both incoming events as well as the state to compute their result. State is therefore required for most operations and transformations performed by the stream processor, e.g. the source operators use state to track processed events while aggregations use state to compute the aggregated value. In contrast to stateless FaaS offerings, state is stored local to the operator processing the events.

Managing local state includes additional operational aspects for the stream processor to handle, such as providing fault tolerance in case of failure and (re)scaling the distributed state when the compute capacity changes [53].

2.7.4. Processing Semantics

Using a checkpointing mechanism of processed results, stream processors can recover from failure and restore managed state. Due to the distributed nature of a stream processor, checkpointing has non negligible overhead which decreases throughput and increases latency for events in the dataflow graph. By relaxing the processing requirements of events, the overhead of checkpointing can be decreased. The following three processing semantics are identified [20]:

At Least Once

At least once guarantees that all messages are at least processed once. This implies that the strategy of dealing with duplicates is left as an exercise to the developer. The process is visualized in Figure 2.5.

At Most Once

At most once processing guarantees that messages are processed at most once. There is no explicit guarantee that when a failure occurs the message has already been processed. The process is visualized in Figure 2.6.

Exactly Once

Exactly once is the combination of at-most-once and at-least-once. As the event stream processor can be a component in a larger system, there are two implementations possible. End-to-end exactly once guarantees that the source and sinks of the messages are taken into account. Local exactly once guarantees only the internal state managed by the event stream processor.

Guaranteeing end-to-end exactly once can be implemented using a two-phase commit algorithm visualize in Figure 2.7.

While these three processing semantics provide constraints and bounds on the processing of events, often the strict constraints of exactly once are not necessary. When the processing of the event is an idempotent operation, processing the same event twice is guaranteed to give the same result. Relaxing the exactly once processing semantics to at least once semantics improves the event stream performance.

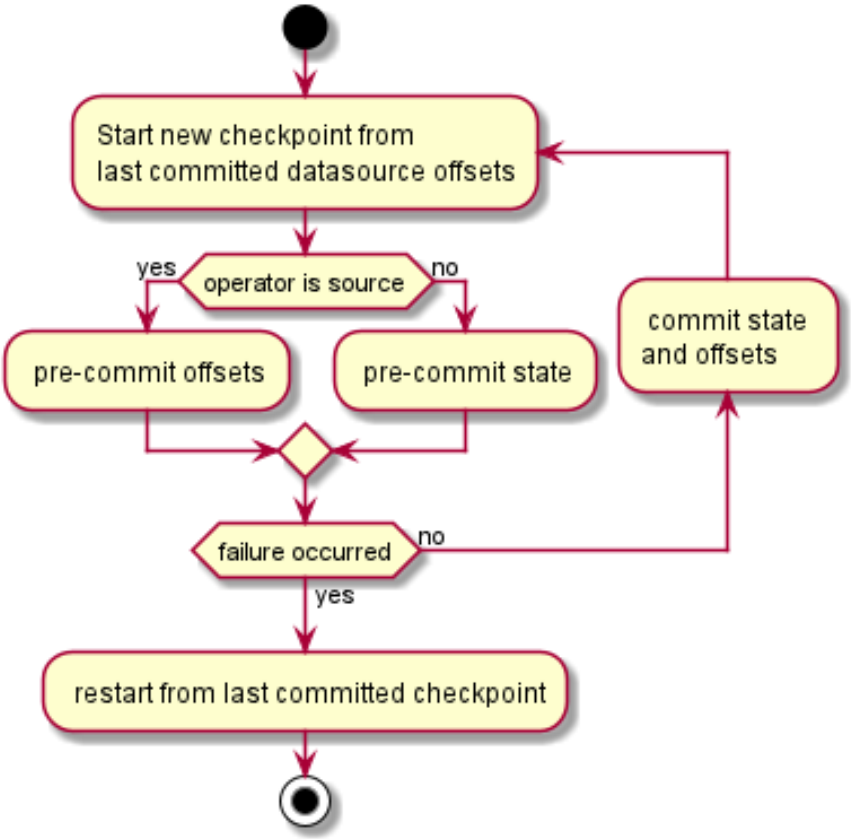


Figure 2.5: At least once event processing

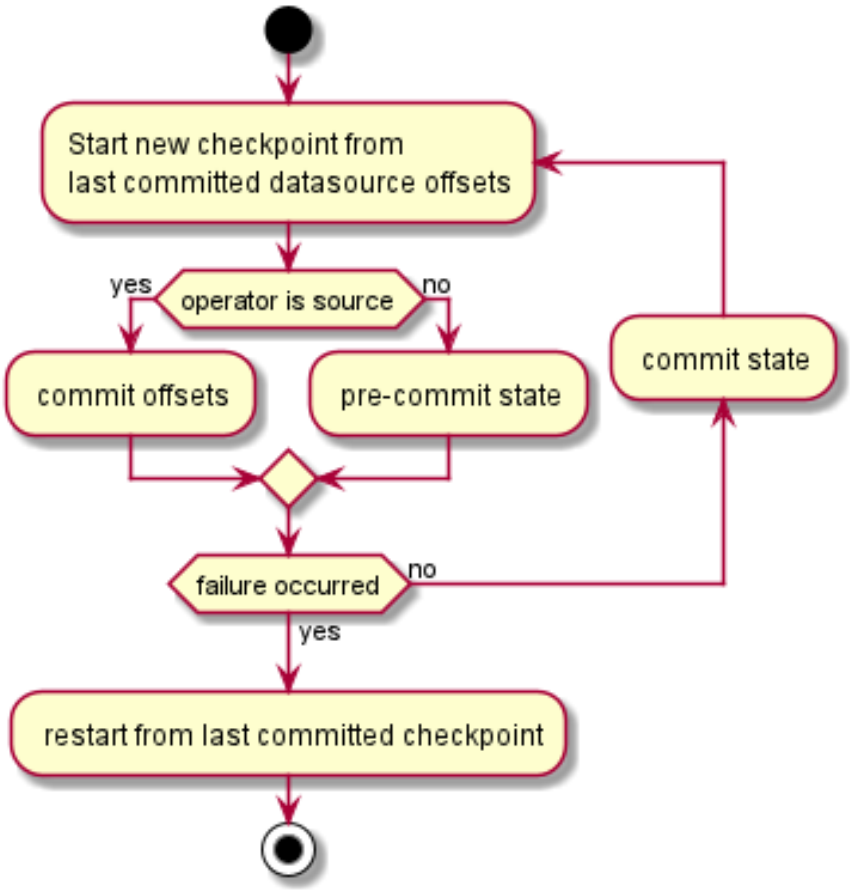


Figure 2.6: At most once event processing

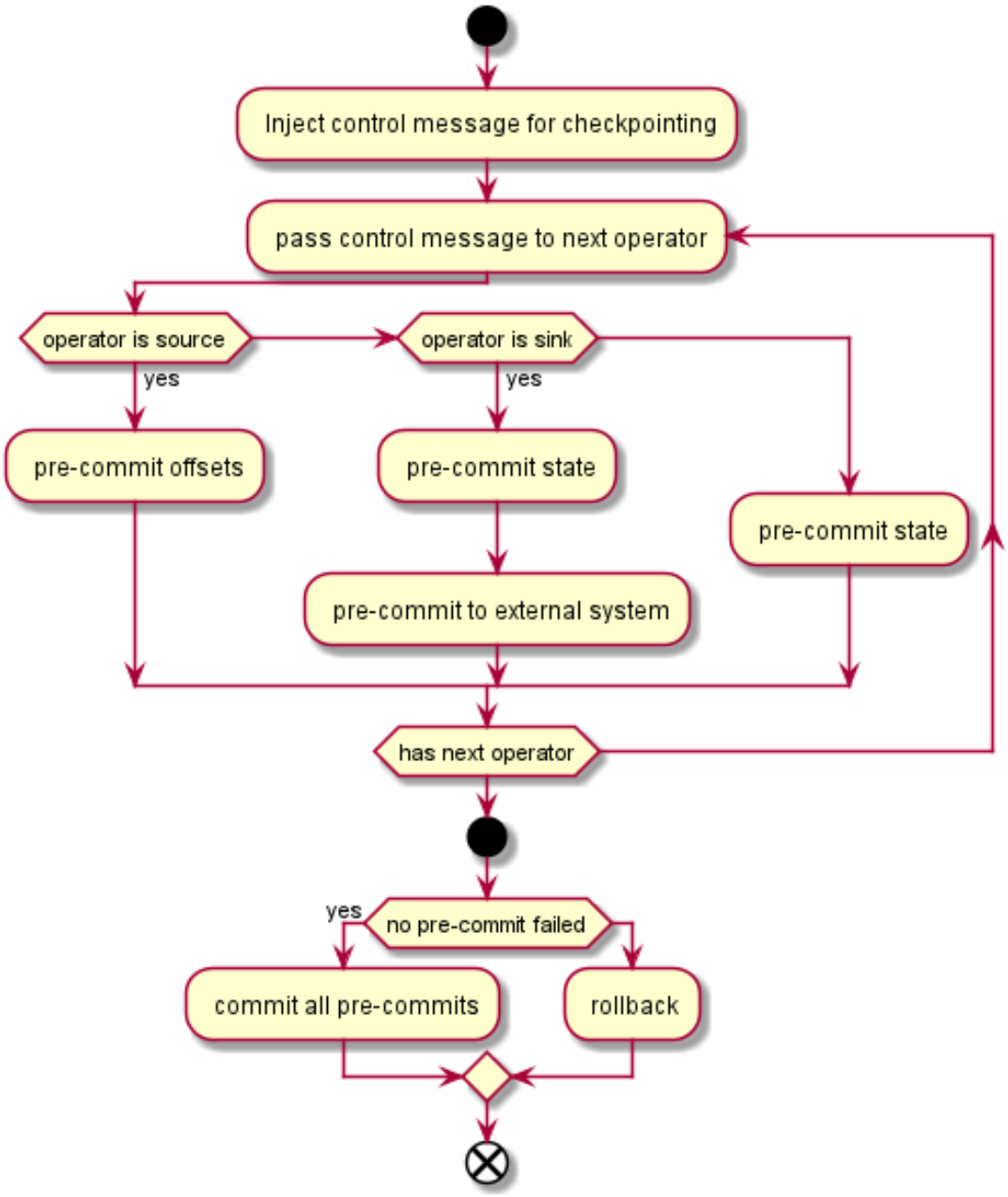


Figure 2.7: Exactly once processing semantics

2.7.5. Apache Flink

Apache Flink [14] is an implementation of a distributed stream processing system. In Apache Flink, a logical graph is equivalent to a dataflow graph. The nodes represent operators which perform some computation on the data stream while the edges represent the passing of data. At runtime, the logical graph is translated into a physical graph. Nodes in the physical graph represent tasks, describing a single parallel instance of an operator [8]. Flink contains a manager component, referred to as the JobManager responsible for:

- **JobMaster —** managing task instances of a physical graph
- **ResourceManager —** managing the allocation of resources for the logical to physical graph transformation
- **Dispatcher —** managing endpoints for dispatching jobs

The resources used by a ResourceManager are managed using TaskManagers. TaskManagers can offer slots as requested by the ResourceManager needed to start a physical graph.

The two main manager services, the JobManager and the TaskManager, illustrate how the distributed nature of Apache Flink can be achieved. A single JobMaster is responsible for many tasks which are managed by TaskManagers. There might be multiple JobMasters on a single JobManager, that depends on the style of deployment.

Environments & Sources

The programming model of Apache Flink uses the concept of a stream execution environment (SEE) to interface with the logical graph of the job. The environment can be dynamically configured to resolve local development and cluster deployments. To create a data stream, at least a single source operator should be added to the SEE. Source operators in the dataflow graph are nodes with no incoming edges and provide communication with external systems such as Apache Kafka to ingest data into the data stream. A list of supported data streams can be found in Table 2.5.

Operators & User Defined Functions

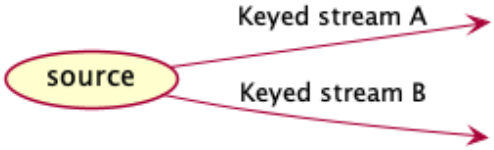
Operations correspond to nodes in the dataflow graph. A complete enumeration of operators is provided in Table 2.7. Operators are often closely related to business logic to be applied on the data stream. An example of such custom processing is an operator which maps incoming numeric events to twice their initial value. An accompanying operator to define the node in the graph is the StreamMap operator. User Defined Functions (UDF) contain the user-code which is executed as part of the operator. Using this programming model, Apache Flink can abstract the physical passing of events and provide a clean public API to use for non-generic use-cases. An enumeration of all UDFs is provided in Table 2.9.

Transformations

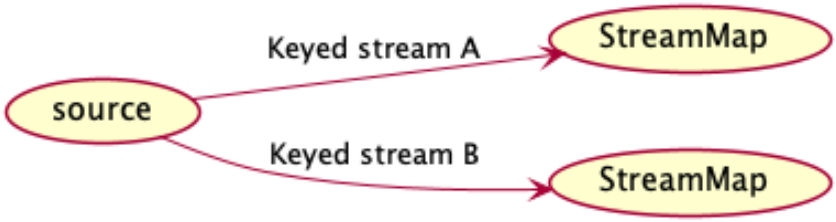
A dataflow graph resembles a directed acyclic graph (DAG) (ignoring iterative streams). The SEE is used to construct the initial source(s) of the dataflow graph. Addition of operators and how these operators pass data is reflected as transformations on the structure of the DAG. An enumeration of transformations in Apache Flink is provided in Table 2.8 along with event passing partitioning strategies in Table 2.6.

An example construction of a dataflow graph using the Apache Flink programming model is provided in Figure 2.8.

(a) use SEE to create source node of DAG



(b) keyBy using the PartitionTransformation



(c) apply OneInputTransformation containing a StreamMap operator with parallelization 2

(d) apply SinkTransformation to complete the DAG

Figure 2.8: Physical dataflow graph

Table 2.5: Different stream types in a stream execution environment

	StreamTypes			
ID	**Name**	**Type**	**Trans.**	**Description**
DATA	DataStream	<T>		Base class representing a datastream
SPLIT	SplitScreen	<T>	SPLIT	Splits the stream into multiple output streams
BROAD	BroadcastConnectorStream	<T,R>		Represents a stream with broadcasted state
KEYED	KeyedStream	<T,K>		Stream partitioned based on presence of a key
CONN	ConnectedStream	<T,R>		A union of two streams of possibly different type
ITER	IterativeStream	<T>	FEEDBACK	Stream that is iterated upon
COGROUP	CoGroupedStream	<T,T2>		Stream containing two inputs
JOINED	JoinedStream	<T,T2>		Two datastreams that have been joined
ALLWINDOW	AllWindowedStream	<T, TimeWindow>		Partitions a non-keyed stream into windows
SINGLE	SingleOutputStreamOperator	<T>		Represents a user defined function on a datastream
WINDOW	WindowStream	<T, K, W>		Partitions a keyed stream into windows

Trans. refers to the transformations as defined in table 2.8

Table 2.6: Partitioner functions used by the physical partitioner mechanism

	Partitioner Functions
ID	**Name**
BROAD	BroadcastPartitioner
SHUFFLE	ShufflePartitioner
FORWARD	ForwardPartitioner
REBALANCE	RebalancePartitioner
RESCALE	RescalePartitioner
GLOBAL	GlobalPartitioner

Table 2.7: Intermediate operators representing business logic on stream types

	Operators
ID	**Name**
MAP	StreamMap
FMAP	StreamFlatMap
PROC	ProcessOperator
FILTER	StreamFilter
PROJECT	StreamProject
TIME	TimeStampsAndWatermarksOperator
CMAP	CoStreamMap
CFMAP	CoStreamFlatMap
CPROC	CoProcessOperator
LKCPROC	LegacyKeyedCoProcessOperator
KCPROC	KeyedCoProcessOperator
COKBROAD	CoBroadcastWithKeyedOperator
COBROAD	CoBroadcastWithNonKeyedOperator
LKPROC	LegacyKeyedProcessOperator
KPROC	KeyedProcessOperator
GREDUCE	StreamGroupedReduce
GFOLD	StreamGroupedFold
SUM	SumAggregator
COMP	ComparableAggregator
REDUCE	StreamGroupedReduce
JOIN	JoinCoGroupFunction

Table 2.8: Transformations which can be applied on the data stream

	Transforms	
ID	**Name**	**Type**
ONE	OneInputTransform	<IN, OUT>
TWO	TwoInputTransformation	<IN1, IN2, OUT>
ABSTR	AbstractMultipleInputTransformation	<OUT>
SOURCE	SourceTransformation	<OUT>
LSOURCE	LegacySourceTransformation	<OUT>
SINK	SinkTransformation	<IN>
UNION	UnionTransformation	<IN>
SPLIT	SplitTransformation	<INOUT>
SELECT	SelectTransformation	<INOUT>
FEEDBACK	FeedbackTransformation	<INOUT>
COFEEDBACK	CoFeedbackTransformation	<F>
PARTITION	PartitionTransformation	
SIDE	SideOutputTransformation	

Table 2.9: User defined functions accompanying an operator

	Functions		
ID	**Name**	**Type**	**Internal Operator State**
AGGR	AggregateFunction	<IN, ACC, OUT>	IN, KEY
BROAD	BroadcastVariableInitializer	<T,O>	
CGROUP	CoGroupFunction	<IN1, IN2, O>	IN1, IN2, KEY
COMB	CombineFunction	<IN, OUT>	
CROSS	CrossFunction	<IN1, IN2, OUT>	
FILTER	FilterFunction	<T>	
FJOIN	FlatJoinFunction	<IN1, IN2, OUT>	
FMAP	FlatMapFunction	<T, O>	
FOLD	FoldFunction	<O,T>	
GCOMB	GroupCombineFunction	<IN, OUT>	
GREDUCE	GroupReduceFunction	<T,O>	
MAP	MapFunction	<T,O>	
MPART	MapPartitionFunction	<T,O>	
REDUCE	ReduceFunction	<T>	
KEY	KeySelector	<IN, KEY>	
PART	Partitioner	<K>	
JOIN	JoinFunction	<IN1,IN2, OUT>	IN1, IN2, KEY
	Streaming Functions		
PROC	ProcessFunction	<I, O>	
CMAP	CoMapFunction	<IN1, IN2, OUT>	
CFMAP	CoFlatMapFunction	<IN1, IN2, OUT>	
CPROC	CoProcessFunction	<IN1, IN2, OUT>	
KCPROC	KeyedCoProcessFunction	<IN1, IN2, R>	
KBPROC	KeyedBroadcastProcessFunction	<IN1, IN2, R>	
BPROC	BroadcastProcessFunction	<KS, IN1, IN2, OUT>	
KPROC	KeyedProcessFunction	<IN1, IN2, OUT>	
WINDOW	WindowFunction	<IN, OUT, KEY, W>	IN, KEY
AWINDOW	AllWindowFunction	<IN, OUT, W>	IN

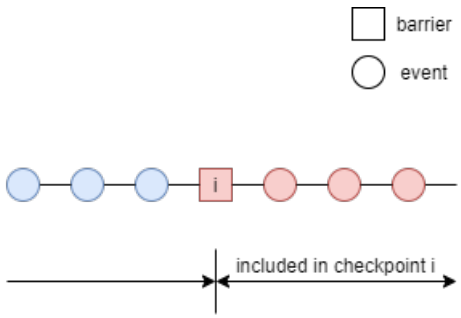


Figure 2.9: Barrier event in datastream

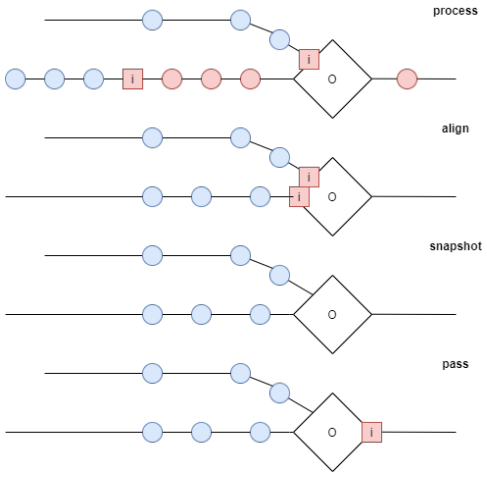


Figure 2.10: Barrier alignment on operator level [7] [25]

Managed State

In subsection 2.7.3, challenges of incorporating state management were discussed. These challenges included:

1. Fault tolerance of managed state
2. Support scaling in a stateful system

Fault tolerance of managed state The stream processor can accumulate vast amounts of state during the lifetime of the job. In case of failure, the state should be recovered without incomplete state manipulations that occurred during the failure. Losing the state can have catastrophic impact. Imagine running a pipeline which depends on a rolling average. Losing past events results in an inability of calculating a correct rolling average. In Apache Flink, fault tolerance of managed state is implemented using a barrier based checkpointing algorithm [25]. Extracting the managed state uses a barrier based extraction algorithm which works similar to checkpointing but using a bottom up methodology starting from the source nodes. Barriers are injected as special control messages in the event stream of the physical graph.

Barrier based mechanisms ensure that all events before the barrier are part of the checkpoint, excluding later events. When an operator accepts multiple inputs, the barriers are first aligned. During alignment, when a barrier arrives at an operator, the incoming edge containing the barrier halts processing until a barrier has arrived at all other incoming edges. Events that are part of an edge for which no barrier has arrived yet are still processed. After a barrier has arrived at all incoming edges, a checkpoint or savepoint is created of the operator's state. The final step is to emit the barrier to all outgoing edges of the operator. The alignment algorithm is visualized in Figure 2.10.

Based on the various processing requirements, various checkpointing intervals can be configured. With at most once processing configured, events that are ingested by the data stream will not be ingested again. With at least once, the events that are not part of a checkpoint are replayed. This might lead to events being processed more than once. Using exactly once processing, each event is checkpointed ensuring that no event is lost or processed more than once. As checkpointing is a resource intensive process, this does lead to lower throughput.

To modify managed state, the Flink State Processing API [82] is used. Using the state processing API, an existing savepoint can be modified by adding new operators or removing existing operators. When an existing operator is modified, the State Processing API requires creation of a new savepoint. Without using the Flink State Processing API, it is possible to deploy new versions of a stateful dataflow graph based on the following rules [11]:

- New stateful operators will be initialized with empty state
- The state of removed operators will also be removed
- Operator I/O can be updated as long as the I/O of operators containing internal operator state (see Table 2.9) does not change

Apache Flink also supports Apache Avro and POJO datatype evolution [10]. Avro types have a predefined set of supported evolutions [5] while POJOs follow the following set of rules:

- Fields can be removed
- New fields can be added
- Existing fields cannot change
- Classname and namespace of POJOs cannot change

Scaling state Rescaling stateless computations can be trivially resolved by adding additional nodes and forwarding events to the new allocated compute capacity or dropping nodes. As discussed in section 1.2, to resemble state in a stateless FaaS architecture, functions resemble state using an external system. The function first initializes by retrieving the state, processes the event and writes back the result to the external system. The drawback of such an architecture is that while compute capacity is trivially scalable, the bottleneck becomes an I/O challenge.

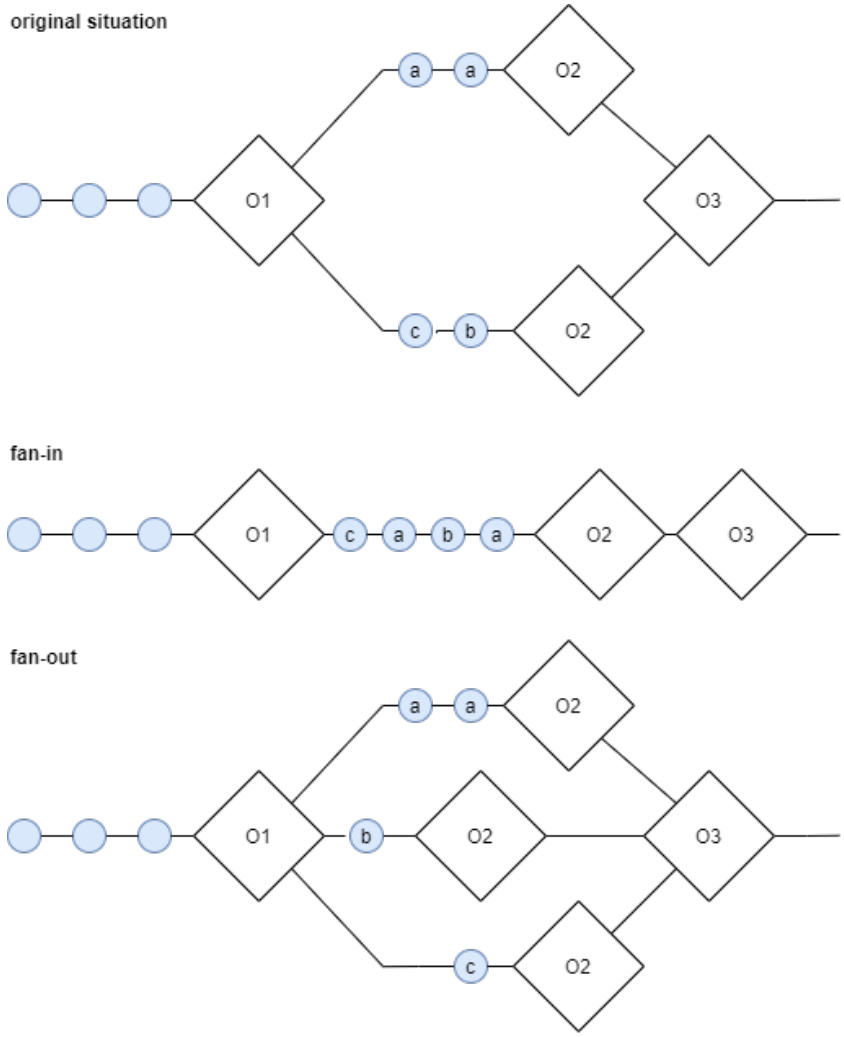
Apache Flink uses the concept of state backends [9] to manage state. State backends are responsible for the physical implementation of persisting state and guaranteeing consistency. For this purpose, three state backends can be distinguished:

- Memory backend — a state backend which stores the state in memory. After checkpointing, the snapshot of the state is send to the JobManager where the state is also stored in-memory. After a failure, the state is lost.
- File system backend — a state backend which persists state to a filesystem.
- RocksdDB backend — a state backend which persists state to a RocksDB database [72]. RocksDB is a key-value store for low latency scenario's. During checkpointing, the RocksDB database will be stored to a persistent file system location (like the file system backend). At runtime, the RocksDB database is locally on the task manager.

As Apache Flink is a distributed stream processor, events might be propagated over a network connection. This requires serialization of the events. The Type Serialization Framework [6] handles the serialization of events and state for persistence and network propagation.

As discussed in subsection 2.7.1, in the physical dataflow graph model, parallel execution is employed to scale the compute capacity. When re-scaling a system either fan-out or fan-in (see Figure 2.11) the state needs to be adjusted correspondingly. Four separate state models can be distinguished [12]:

Figure 2.11: Rescaling a dataflow graph fan-in or fan-out



1. Operator state — state local to the operator
2. Broadcast state — special case of operator state where state is broadcasted to all parallel downstream operators
3. Keyed state — state associated with a key
4. Window state — state associated with a window

To rescale a dataflow graph, the checkpointing algorithm is used to restore the graph. Operator state is stored as a list and divided over the parallel operators [13]. Broadcast state uses a similar pattern where the assumption is that all parallel operators share the same state, hence identical state is redeployed to all parallel operators. Keyed state can be employed on a keyed stream. Keyed streams allow for efficiently parallelizing the dataflow graph. During a keying operation, the stream is converted to a keyed stream by deterministically deriving a key. All events that are associated with a specific key are sent to the same operator. Keyed state is therefore by definition sharded and processed by a subset of the available parallel operators. During the rescaling, the state of an operator must be the state that is associated with the events that the operator is assigned to process [13].

There are two interfaces that provide state access: RichFunction and CheckpointedInterface. Respectively, an interface that provides access to the runtime context which in turn provides access to state, and an interface that provides access to operator state or keyed state (if applied on a keyed stream). A high level overview of acquiring these state handles is visualized in Figure 2.12.

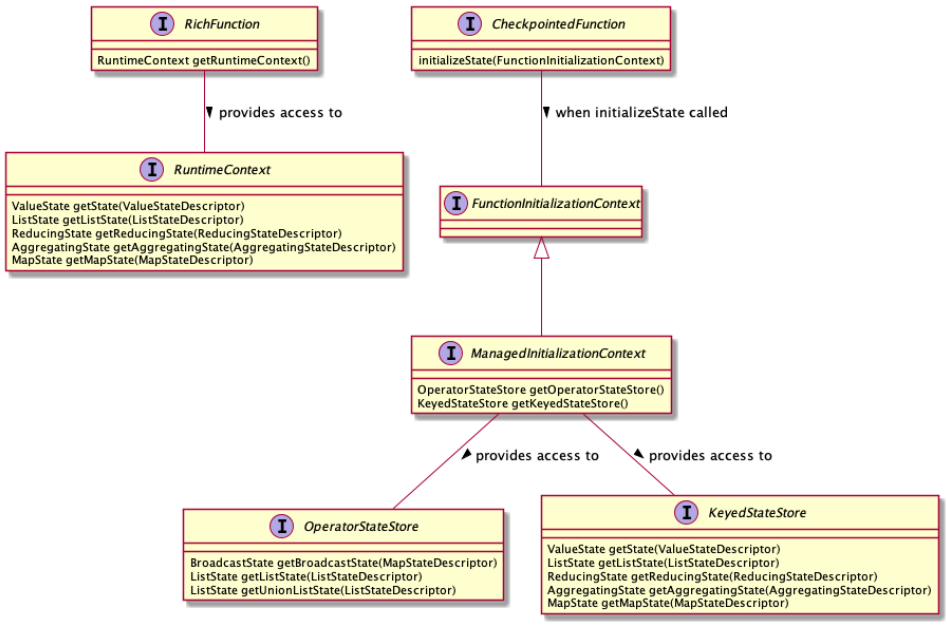


Figure 2.12: Visualization of state access

The gateways which implement access to the state are referred to as state descriptors. During the described lifecycle hooks of CheckpointedInterface and RichFunction, the state descriptors can register themselves for access to specific state. Apache Flink offers five concrete state descriptors for various use cases:

1. ValueStateDescriptor for persisting a single value
2. ListStateDescriptor for persisting a list of values
3. MapStateDescriptor for persisting a key-value pair
4. AggregatingStateDescriptor for persisting the aggregated result of values

5. ReducingStateDescriptor for persisting the reduced value of values

API The public API for applying transformations on the DAG is enumerated in Table 2.10. The table references previously reported tables on stream types, operator types, transformations, and partitioner. When a column is not defined, this implies that no implicit transformations are applied by Flink. Implicit behavior such as the existence of some user functions add internal operator state to the dataflow graph. This operator state imposes additional complexities on the evolution of the DAG in later stages of its lifecycle.

Table 2.10: public API of Apache Flink datastream

DataStream	**API**					
Name	**Input**[a]	**Output**[a]	**Input**[b]	**Operators**[c]	**Transformation**[d]	**Partitioner**[e]
union	DATA	DATA	ST.DATA	CMAP		
split	DATA	SPLIT	OutputSelector	CFMAP*		
connect	DATA	CONN	ST.DATA			
connect	DATA	BROAD	ST.BROAD			
keyBy	DATA	KEYED	F.KEY			
partition	DATA	DATA	F.PART		PARTITION	
broadcast	DATA	DATA			PARTITION	BROAD
broadcast	DATA	BROAD	MapStateDescriptor		PARTITION	BROAD
shuffle	DATA	DATA			PARTITION	SHUFFLE
forward	DATA	DATA			PARTITION	FORWARD
rebalance	DATA	DATA			PARTITION	REBALANCE
rescale	DATA	DATA			PARTITION	RESCALE
global	DATA	DATA			PARTITION	GLOBAL
iterate	DATA	ITER			FEEDBACK*	
map	DATA	SINGLE	F.MAP	MAP	ONE	
flatMap	DATA	SINGLE	F.FMAP	FMAP	ONE	
process	DATA	SINGLE	F.PROC	PROCESS	ONE	
filter	DATA	SINGLE	F.FILTER	FILTER	ONE	
project	DATA	SINGLE		PROJECT	ONE	
coGroup	DATA	COGROUP	ST.COGROUP			
join	DATA	JOINED	ST.DATA			
timeWindowAll	DATA	ALLWINDOW				
countWindowAll	DATA	ALLWINDOW				
windowAll	DATA	ALLWINDOW	WindowAssigner			
assignTimestampAnd Watermarks	DATA	SINGLE	WatermarkStrategy	TIME		
select	SPLIT	DATA			SELECT	
keyBy	CONN	CONN				
map	CONN	SINGLE	F.KEY	CMAP	TWO	
flatMap	CONN	SINGLE	F.CMAP	CFMAP	TWO	
process	CONN	SINGLE	F.CFMAP	CPROC	TWO	
process	CONN	SINGLE	F.CPROC	LKCPROC	TWO	
process	CONN	SINGLE	F.KCPROC	KCPROC	TWO	
process	BROAD	SINGLE	F.KBPROC	COKBROAD	TWO	
process	BROAD	SINGLE	F.BPROC	COBROAD	TWO	
process	KEYED	SINGLE	F.CPROC	LKPROC		
process	KEYED	SINGLE	F.KPROC	KPROC		
timeWindow	KEYED	WINDOW				
countWindow	KEYED	WINDOW				
window	KEYED	WINDOW	WindowAssigner			
reduce	KEYED	SINGLE	F.REDUCE	GREDUCE	TWO	
fold	KEYED	SINGLE	F.FOLD	GFOLD	TWO	
sum	KEYED	SINGLE		SUM		
min	KEYED	SINGLE		COMP		
max	KEYED	SINGLE		REDUCE		
minBy	KEYED	SINGLE		JOIN		
maxBy	KEYED	SINGLE		JOIN		
aggregate	KEYED	SINGLE	F.AGGR	JOIN		
with	JOINED	JOINED				
equalTo	JOINED	JOINED				
window	JOINED	JOINED				
apply	JOINED	SINGLE	F.JOIN			
apply	JOINED	SINGLE	F.FJOIN			
apply	JOINED	DATA	F.CGROUP			
with	COGROUP	COGROUP				
equalTo	COGROUP	COGROUP				
window	COGROUP	COGROUP				
apply	COGROUP	DATA	F.CGROUP			

* marks transitive transformation due to construction of StreamType, see table 2.5

[a] references ID of stream type, see table 2.5, [b] ST references StreamType (see a), F references UDFs (see table 2.9), [c] references ID of operator, see table 2.7, [d] references ID of transformation, see table 2.8, [e] references ID of partitioner, see table 2.6.

3

Stateful Dataflow Graph Evolution

This chapter will describe the implementation of the stateful dataflow graph evolution process. In section 3.1, the general outline of the entire process is discussed. The entire process also describes the invocation of the job evolution process as well as subsequent deployments performed to complete the process. The remaining sections will describe different phases of the evolution process.

3.1. Outline of the Stateful Dataflow Graph Evolution process

Assume that there is a running streaming job[1] J_1 with operators $O_{J_1} = \{o_1, o_2, o_3, ..., o_n, o_{n+1}\}$. Operators may or may not use local state. Given some other streaming job J_2 where $O_{J_1} \neq O_{J_2}$ using equality based on the operator unique identifier and the attributes of the operator. It is assumed that even though $O_{J_1} \neq O_{J_2}$, there exists some transformation $f(J) \rightarrow J^*$ that when applied on J_1: $f(J_1) \rightarrow J_2$ will ensure $O_{J_1} = O_{J_2}$.

The general outline invokes a trigger TRGR which *triggers* the evolution process. The evolution process is agnostic to the type of trigger used. The evolution process starts by deriving a schema representing the nodes of the DAG. Defining such a schema S bounds the attributes in scope of the evolution process and allows a potentially unbounded job definition J to be bounded and compared. With a source schema S_1 and a target schema S_2, the difference in schema can be referred to as $S_{\text{diff}} = \{d_1, d_2, d_3, ..., d_n, d_{n+1}\}$. For the evolution process to complete, no differences should remain: $S_{\text{diff}} = \emptyset$. To achieve an empty S_{diff}, the source schema is incrementally transformed using a set of transformations $T = \{t_1, t_2, t_3, ..., t_n, t_{n+1}\}$ by using some transformation function $trans(S^i, t_i) \rightarrow S^{i+1}$. The transformations are applied iteratively and can be represented as $S^* = \sum_{i=0}^{N} trans(S^i, t_i)$, where $S^0 = S_1$, the source schema. The set of transformations can be either user provided (explicit) or derived by the schema differences S_{diff} (implicit). Assuming that S_{diff} is $\emptyset$, the transformations are applied to the underlying physical state of J_1.

The evolution can now be deployed. The evolution process is agnostic to the deployment process, for example the methodology discussed by Bartnik et al. [20] could be used as a delivery method for live reconfiguration. The evolution process is demonstrated using an Apache Flink native deployment method comprising of a *stop-and-restart* delivery using a transformed *savepoint*.

The outline consists of several distinct phases: the trigger phase containing the invocation of the process, the schema extraction phase where the schema of the job is extracted, the diffing phase where two schemas are *diffed* to find the transformations needed to resolve any differences, the migration phase where the state is migrated, and a deployment phase to deploy the evolved job.

[1] Note that a streaming job represents a dataflow graph, which can be modeled as a directed acyclic graph (DAG)

Trigger phase

When some TRGR arrives, the context of the evolution needs to be acquired. The context relates to the prerequisites of the evolution process which are to be validated before starting the evolution process. The TRGR can be said to act as a gateway where incorrect contexts are aborted before start. Until the diffing phase, only the software artifacts containing J_1 and J_2 are required. The complete evolution process also containing the migration phase and deployment phase requires a running instance of J_1 in order to migrate any incompatible state during $f(J_1) \rightarrow J_2$. As stated in section 3.1, a HTTP based trigger phase is implemented as detailed in section 3.2.

Schema Generation phase

Using the software artifacts for J_1 and J_2, their schema representation should be derived, respectively S_1 and S_2. The schema representation bounds the potentially unbounded set of attributes that can represent the DAG. As handles for local state are registered at runtime, a new type of execution context is introduced which proxies method invocations to the native execution context. During the invocation of proxy method, the request made by the callee is stored for retrieval by the schema generation algorithm. The graph transformations [2] that are supplied to construct the DAG are *replayed* in an *abstract* DAG context to build the schema.

Diffing phase

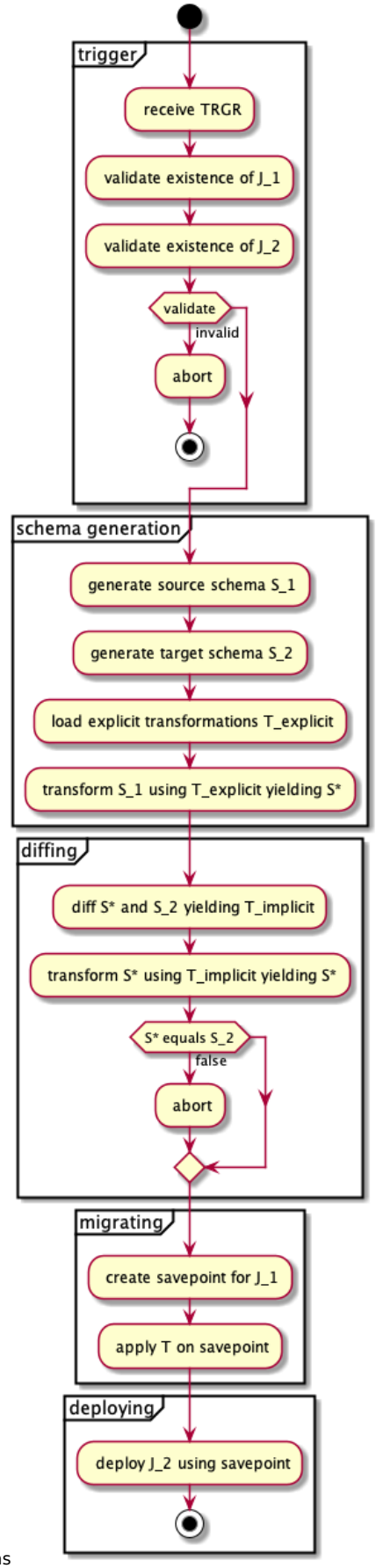
Having two schemas S_1 and S_2, for the evolution process it is imperative that both versions of the schema are compared to derive differences. As the schema contains attributes with various data types, a diverse set of equality operations should be used to determine the type of difference of a specific attribute. This yields an evolution type, e.g. addition or removal, which drives the schema transformation and schema analysis components to resolve the evolutions by describing a set of transformations. This phase can thus either yield a complete list of transformations resulting in an empty S_{diff} or abort the evolution process as the schemas are incompatible.

Migration phase

During the migration phase, a global state view is acquired from the running job J_1. The schema transformations are applied on the global state to perform state migration.

Deployment phase

For demonstration purposes, an *all-at-once* deployment is used which cancels the running job J_1 (during the acquisition of the global state) and starts the target job J_2 hydrated with the migrated state. The evolution process can be adapted to use different migration and deployment strategies.



[2] Note that graph transformations are not equivalent to schema transformations

3.2. Trigger Phase

The component processing TRGR events acts as a provider of the evolution process functionality. Using an HTTP based delivery method, the component processing the TRGR events can be referred to as an endpoint. Consumers who conform the public API of the endpoint should be notified of any end state that the evolution process may reach. This includes:

1. Prerequisites of the TRGR not being met
2. Failures due to errors in user supplied software artifacts
3. Failures due to errors in the evolution process
4. Successful evolution of the streaming job

HTTP endpoints can be triggered using libraries such as cURL [27] or GNU Wget [91]. Minimal container images can be found on public repositories [3] with over a billion downloads. These publicly provided images enable trivial communication between the continuous deployment processes as discussed in section 2.2 and the evolution endpoint.

The endpoint acts as a governor of the evolution process. This implies that the endpoint drives the invocation of the phase (or sub steps) when a stage has finished. In order to govern the process, the endpoint needs to be able to derive the following three values:

1. A reference to the source software artifact location J_1 and the arguments used to start J_1
2. A reference to the target software artifact location J_2 and the arguments used to start J_2
3. A reference to a running instance of J_1

If any of these values is unknown, the governor cannot be constructed and the endpoint notifies the consumer that the operation is aborted.

3.3. Schema Generation Phase

Reaching the schema generation phase implies that for both the source and target version of the streaming job, there is a reference to the source software artifact location and its corresponding arguments. As described in section 2.7, a logical dataflow graph corresponding to job J can be represented as a directed acyclic graph (DAG). Directed graphs are pairs of $(N, E \subseteq N \times N)$ where N is a node in the graph and E is a directed edge [34]. The DAG features a source node which emits events through the graph and a sink for events leaving the graph.

While constructing a streaming job, the underlying model is a set of transformations $T = \{t_1, t_2, t_3, ..., t_n, t_{n+1}\}$ to be iteratively applied on an empty graph G^0 that yields the logical graph G^{n+1}. The nodes in the graph contain a unique identifier which provides a stable reference for redeployments. When some graph G is restarted, the nodes are hydrated with state related to their unique identifiers.

In subsection 2.7.5, the various types of operators and user functions are described. These operators relate to the specific implementation of Apache Flink but describe use-cases that are generically applicable to other stream processing systems. The operators represent nodes in graph G. To generalize the possible nodes, the list of attributes collected for a node is specified in Table 3.1. A node which is generalized to this set of attributes can be referred to as an *abstract node*

To extract non-state related attributes, transformation specific extractions are executed using the transformations described in Table 2.8. Implicit operator state (see Table 2.10) is extracted with an identical strategy as non-state attributes. State attributes are registered at runtime. To extract these attributes, an extraction context is introduced which shadows the default context used to initialize the state and the UDF. The extraction context is responsible for storing all method invocations on the shadowed target. The extraction context can then be queried for state attributes. The process of acquiring state handles is visualized in Figure 3.1.

[3] https://hub.docker.com/r/curlimages/curl

Table 3.1: Attributes collected for a generic node N in graph G

Attribute	Optional	Description
id	no	The unique identifier of the node
output type	no	The data type emitted to downstream nodes
transformation	no	The transformation used to transform the logical dataflow graph
name	yes	The human readable name of the node
input type	yes	The data type of incoming events
secondary input type	yes	The data type of a node receiving ≥ incoming edges
operator state	yes	State handles describing operator state
window state	yes	State handles describing window state
keyed state	yes	State handles describing keyed state
key type	yes	The data type of the key used to partition the stream
key selector	yes	The UDF to derive the key
operator	yes	The operator introduced to the graph
trigger	yes	Trigger used for window operators
watermark strategy	yes	The watermark strategy for timing events
evictor	yes	The evictor used for window operations
window assigner	yes	The assigner used for window operations
user function	yes	The UDF used by the operator
max parallelism	yes	The maximum amount of parallel operators

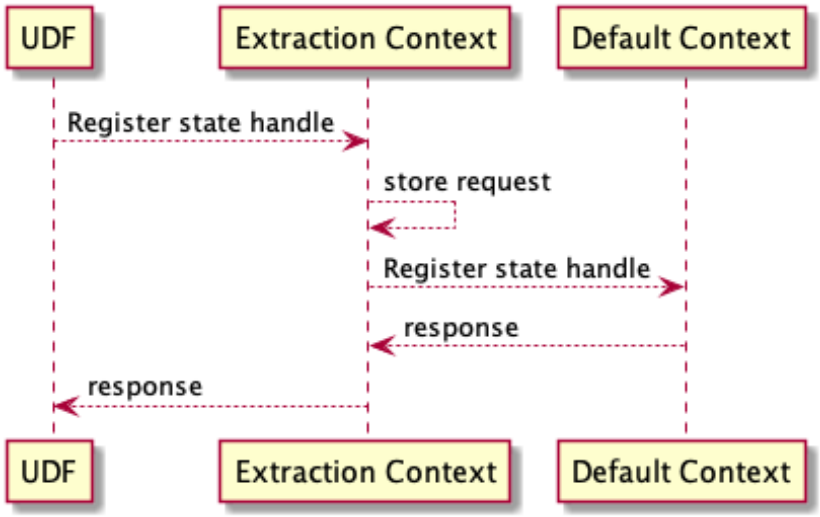


Figure 3.1: Extracting state from UDFs

Processing all transformations T that would yield G^{n+1} through the extraction process, yields *abstract* nodes for each $N \in G$. The edges do not relate to physical processing of events and the functionality of the edges is captured in the attributes extracted in *abstract* operators. Dropping the edges yields an underlying model of the schema S as a set of *abstract* operators.

3.3.1. Transforming nodes in the schema

It is assumed that the generated schemas S_1 and S_2, for respectively J_1 and J_2, differ in some *abstract* node with unique identifier i: $N_1^i \neq N_2^i$ [4]. To make progress towards the desired end result of $S_{\text{diff}} = \emptyset$, N_1^i is to be transformed to better match N_2^i. As each transformation on the source schema makes progress towards $S_{\text{diff}} = \emptyset$, repeatedly applying transformations eventually converges at $S_{\text{diff}} = \emptyset$.

[4] If no such node exists, the schemas are equivalent at which point the evolution process can simply stop

A transformation t features a predicate for the source schema and target schema which uniquely specifies the *abstract* node and its corresponding attribute. The second feature of a transformation t relates to the actual mutation on the schema. A mutation might introduce new values for attributes, update an existing value or remove an attribute value. An example transformation where a broadcast state handle data type is changed from {long, long} to {string, string} can be seen in Listing 3.1.

Listing 3.1: state migration of broadcast state

```
val s = new TupleTypeInfo[Tuple2[Long, Long]](of(classOf[Long]), of(classOf[Long])) // source type
val t = new TupleTypeInfo[Tuple2[String, String]](of(classOf[String]),of(classOf[String])) // target type
val migration = StateMigration.broadcast()
      .setSource(CATEGORIZE_BIDS_UID, BROADCAST_NAME)
      .setTarget(CATEGORIZE_BIDS_UID, BROADCAST_NAME)
      .map(s, t,(m: Tuple2[Long, Long]) => new Tuple2(m.f0.toString, m.f1.toString))
```

3.4. Diffing Phase

The attributes collected in Table 3.1 can be interpreted as a set of key-value pairs (K, V) where K is the attribute name and V the attribute value. When provided with two sets of (K, V) pairs, the underlying difference in schema can be represented by three change primitives: addition, removal, and update. Consider the following mapping:

$$f_{source} : K_{source} \rightarrow V_{source} \tag{3.1}$$
$$f_{target} : K_{target} \rightarrow V_{target} \tag{3.2}$$

Additions are key-value pairs for some key that exists in f_{target} but not in f_{source}. Removals are key-value pairs for some key that exists in f_{source} but not in f_{root}. Updates are key-value pairs for keys that are identical between f_{source} and f_{target} but contain different values:

$$C_{\text{addition}} = \{(k, f_{target}(k)) : \forall k \in K_{target} \wedge k \notin K_{source}\} \tag{3.3}$$
$$C_{\text{removal}} = \{(k, f_{source}(k)) : \forall k \in K_{source} \wedge k \notin K_{target}\} \tag{3.4}$$
$$C_{\text{update}} = \{(k, f_{source}(k), f_{target}(k)) : \forall k \in K_{source} \wedge k \in K_{target} \wedge f_{source}(k) \neq f_{target}(k)\} \tag{3.5}$$

The set of C_{update} can be non-trivial to determine. Equality checks for functions can be based on an abstract syntax tree (AST) comparison but in general would require solving the halting problem which is proven to be algorithmically unsolvable [23]. Updates on non-primitive data types are assumed to be handled using a data structure evolution mechanism. An example data format that implements these evolution mechanisms is Apache Avro [90], supported by major event streaming platforms such as Apache Flink, Apache Spark, and Apache Kafka

The change primitives are captured in either unary evolutions or binary evolutions. Unary evolutions rely on a single schema to capture the change while binary evolutions require values from both a source and target schema. Finally, three additional *meta* change types are introduced that simplify the evolution process: unchanged, unknown, and move. The evolution types are listed in Table 3.2.

3.4.1. Deriving (implicit) transformations for schema compatibility

Note the two previously defined types of transformations to be applied to the schema from section 3.1: explicit and implicit. In Listing 3.1, an example of an explicit transformations is shown.

The source schema S_1 is transformed using the set of explicit transformations $T_{\text{explicit}} \subseteq T$. It is assumed that for the remainder of this section, $T_{\text{explicit}} \subset T$ such that $T_{\text{implicit}} \neq \emptyset$. To derive the transformations, the various types of evolution scenarios needs to be defined. Common evolution scenarios are defined by Jong, Deursen, and Cleve [51] and can be found in Appendix A. The scenarios are mapped to a streaming state model:

Table 3.2: Evolution types

Type of Evolution	Type of Change	Description
Unary	Addition	Key that exists in target schema but not in source
Unary	Removal	Key that exists in source schema but not in target
Unary	Unchanged	(k, v) pair that is identical in source and target
Unary	Unknown	Change on non-comparable type
Binary	Changed	Different value for identical keys in source and target
Binary	Moved	Same value for different keys in source and target [5]

- **S1:** Adding a state handle to an operator stateless
- **S2:** Renaming a state handle stateful
- **S3:** Dropping a state handle stateless
- **S4:** Modifying the datatype of a state handle stateful
- **S5:** Renaming an existing operator stateful

Note that scenarios featuring foreign keys, nullable values or default values are excluded from the listing. These are features specific to the persistence model used by Jong, Deursen, and Cleve [51].

Two additional common scenarios related to stream processing services are identified by Bartnik et al. [20]:

1. **S6:** Introduction of new operators stateless
2. **S7:** Changing the operator function stateless

Based on empirical usage, the following additional scenarios are identified:

1. **S8:** Removal of an operator stateless
2. **S9:** Changing the key selector function stateful

The scenarios tagged as stateless can often be captured using unary evolutions. State handles are registered using UDFs, as the UDF is part of the source and target schema, the evolution can be implicitly resolved. The introduction and/or removal of new operators relates to the propagation of events through the DAG which can implicitly be resolved based on the schema. This holds for both a *stop-and-restart* deployment as well as a *pause-and-resume* deployment as demonstrated by Bartnik et al. [20].

Renaming a state handle (*S2*) and renaming an existing operator (*S5*) are binary move evolutions. This is a special case of a unary addition evolution and a unary removal evolution. This special relationship between two unary evolutions can only be explicitly defined.

Modifying the data type of a state handle can be handled implicitly based on the source data type and the target data type. When both types are primitives an implicit mapping $f(s) \rightarrow t$ can be defined which migrates physical state from the source data type to the target data type. Note that although such a mapping can exist between two data types, the underlying physical state should be compatible with the expected mapping input format. For example, a numerical value stored as a list of characters can be mapped to a numerical data type based on the assumption that no single character is of a non-numerical value. A full table containing mappings is shown in Table 3.3.

Updating the key selector might resolve in the key being of a changed data type. The state persisted scoped to a specific key needs to be *re-keyed* based on the new key selector output type. The *re-keying* can be resolved implicitly based on the mappings provided in Table 3.3 or be explicitly supplied.

Table 3.3: Data type mappings

	Datatype Mappings	
soure	**target**	**mapping**
short	int,long,float,double	cast
short	boolean	(v: short): boolean -> v == 1
short	String	(v: short): String -> String.valueOf(v)
int	short,long,float,double	cast
int	boolean	(v: int): boolean -> v == 1
int	String	(v: int): String -> String.valueOf(v)
long	short,int,float,double	cast
long	boolean	(v: long): boolean -> v == 1
long	String	(v: long): String -> String.valueOf(v)
float	short,int,long,double	cast
float	boolean	(v: float): boolean -> v == 1.0
float	String	(v: float): String -> String.valueOf(v)
double	short,int,long,float	cast
double	boolean	(v: double): boolean -> v == 1
double	String	(v: double): String -> String.valueOf(v)
boolean	short	(v: boolean): short -> (v == true) ? 1 : 0
boolean	int	(v: boolean): int -> (v == true) ? 1 : 0
boolean	long	(v: boolean): long -> (v == true) ? 1 : 0
boolean	float	(v: boolean): float -> (v == true) ? 1.0 : 0.0
boolean	double	(v: boolean): double -> (v == true) ? 1.0 : 0.0
boolean	String	(v: boolean): String -> String.valueOf(v)
String	short	(v: short): boolean -> Short.parseShort(v)
String	int	(v: short): boolean -> Integer.parseInt(v)
String	long	(v: short): boolean -> Long.parseLong(v)
String	float	(v: short): boolean -> Float.parseFloat(v)
String	double	(v: short): boolean -> Double.parseDouble(v)
String	boolean	(v: short): boolean -> Boolean.parseBoolean(v)

Provided the source schema S_1 and target schema S_2 with some set of explicit transformations defined where $T_{\text{explicit}} \subset T$. Transforming the source schema using T_{explicit} yields S_{explicit}. Given that $S_{\text{explicit}} \neq S_2$ it must hold that:

$$(C_{\text{addition}} \neq \emptyset) \vee (C_{\text{removal}} \neq \emptyset) \vee (C_{\text{update}} \neq \emptyset)$$

Categorize a change $\{c : c \in C_{\text{addition}}$ or $c \in C_{\text{removal}}$ or $c \in C_{\text{update}}\}$ based on scenario S1-S9 to a transformation t_c. Apply the transformation to the schema using $trans(S^i, t_c) \rightarrow S^{i+1}$. The process repeats by *re-diffing* the new schema S^{i+1} with S_2 yielding new C_{addition}, C_{removal} and C_{update} for which a change can be categorized and processed.

The process of deriving these transformations halts when:

$$(C_{\text{addition}} = \emptyset) \wedge (C_{\text{removal}} = \emptyset) \wedge (C_{\text{update}} = \emptyset)$$

Implying that no changes are left to process and $S^{i+1} = S_2$. The following is intermediate output of the evolution process when $S^{i+1} = S_2$. The example shows an execution plan with the changes in schema identified between the source and target version [6]:

```
# transformations.KeyedBroadcastStateTransformation(categorize-bids) {
    key type     : Long
  ? key selector : DataStream$$anon$2
    maxParallelism : -1
    types {
        input        : PojoType<Bid>
        output       : Java Tuple2<Long, Long>
    }
    broadcast state {
    ~ categories : MapStateDescriptor[Map<Long, Long>] -> MapStateDescriptor[Map<String, String>]
    }
}

# operators.StreamMap(average-price) {
  ~ user function : AveragePriceCategory -> AveragePriceCategoryV2
  ~ key type     : Long -> String
  ? key selector : DataStream$$anon$2
    maxParallelism : -1
    types {
        input        : Java Tuple2<Long, Long>
        output       : Java Tuple2<Long, Double>
    }
    keyed state {
        count : ValueStateDescriptor[Integer]
        average : ValueStateDescriptor[Double]
    }
}


Applying the following explicit migrations:
    [0] - MapStateMigration
        categorize-bids.categories[Java Tuple2<Long, Long>] ->
        categorize-bids.categories[Java Tuple2<String, String>]
Applying the following implicit migrations:
    [1] - MapKeyTypeMigration average-price -> average-price
```

[6] Classes are shortened to only show the class name in the example, no package information or additional attributes.

3.5. Migration phase

During migration, the transformations $T = T_{\text{explicit}} \cup T_{\text{implicit}}$ are used to (re)-process the managed state. The performed work builds on a *stop-and-restart* deployment mechanism. With such a mechanism, processing the underlying physical state is sufficient for the evolution process. Previous work [20] has shown that injecting special events in the event stream can be used for live reconfiguration using *pause-and-resume* deployments. The migration phase assumes a global state view V for J_1.

Reflect on the running streaming job modeled as a set of operators $O = o_1, o_2, o_3, ...o_n$. With a *stop-and-restart* deployment, the state migration can occur in an offline scenario. A strategy for migrating the state of a single operator $o \in O$ can be repeated until all operators have successfully migrated.

The state migration process can be modeled as a special DAG $J_{\text{processor}}$ where the event propagation (including the keying and partitioning) is mimicked from the original and the nodes are operators containing mapping functions. Replaying the state contained in the source job J_1 through $J_{\text{processor}}$ yields the new state which can be used for the *stop-and-restart* deployment. Apache Flink provides a state processing API which is used as a runtime for the state migration.

To construct an operator o for $J_{\text{processor}}$, acquire the source operator o_1 from S_1 and the target operator o_2 from S_2. The state handles from o_1 can be referred to as H. These state handles relate to the state stored for the operator including operator state, keyed state, window state, and broadcast state. Acquire all transformations $T_{o_1} = \{t \in T | o_{1_{uid}} = t_{uid}\}$. Apply the transformations T_{o_1} on H:

1. transformation for S3: dropping a state handle: *remove the state handle from H*
2. transformation for S1, S2, S4-S9: *H does not change*

Using H, acquire a dataset of events from V specific to the operator o: V_o. If T_{o_1} contains a transformation for S9 (changing the key selector), map the acquired keyed events to the new data type. Use V_o to replay events on a *mini* DAG specific to operator o. The operator o contains the same unique identifier as o_2 (such that when J_2 is started, the corresponding state is properly hydrated). o contains the mapping functions to process S4 (changing the data type of a state handle).

This process is repeated for all operators in O yielding $J_{\text{processor}}$. $J_{\text{processor}}$ is executed in an *offline* context as a bounded stream (note that the state of J_1 is finite). Once $J_{\text{processor}}$ finishes, the state is properly migrated to match the schema S_2. The migrated state can be referred to as V_{migrated}.

3.6. Deployment phase

During deployment of J_2, attach the migrated state V_{migrated}. The various deployment strategies discussed in section 2.3 can be used for the redeployment of J_2. The performed work builds on an *all-at-once* deployment model where J_1 is stopped during the creation of V. Once V_{migrated} is ready, J_2 is started. If the deployment of J_2 fails, J_1 can be restarted with V. The *all-at-once* migration complies with *exactly-once* processing semantics as no new input can cross the checkpointing barrier introduced in the event stream during the creation of V [7].

3.7. Limitations of the stream evolution process

The performed work builds on the State Processing API provided by Apache Flink. Limitations of the State Processor API include:

- Broadcast state must fit into memory of the task manager responsible for processing the state migration
- Only new operators can be added to an existing savepoint

This implies that updating the schema of an operator results in total state migration. Stream processors can run for long periods of time and accumulate terabytes of state. Total (re)-processing

[7] Note that the checkpointing, savepointing and restart with savepoint mechanisms already exist in Apache Flink. The demonstrated deployment phase uses these existing mechanisms for deployment

of state can take a long time to process. When part of the schema changes, all underlying state impacted by the schema evolution is re-processed. Mehta, Spooner, and Hardwick [57], discussed in section 2.4, mention that state should not be re-processed upon evolution of the schema. Their criteria are based on the incompatibility between shared objects. The proposed notion of dynamically modifying the stored objects [57] can be used to avoid the need for total re-processing of state but rather re-process on demand. This approach does suffer a runtime performance cost.

Because the state migration uses an *all-at-once* approach, there is no intermediate schema representation that can be queried. This implies that transformations which rely on previous transformations in the same evolution process should be applied consecutively. Database management systems execute statements immediately or virtually if part of a transaction. Subsequent statements use the modified version of the schema.

The evolution process relies on explicit transformation to be available to handle binary evolution scenarios. When evolving a data structure, an evolution mechanism such as Apache Avro [90] can be used. *Implicit* transformations could also be derived from mappings on the data structure level based on:

1. Generic serialization scheme such as $\mathrm{toString}$
2. Generic builder pattern such as public <target> from<source>$(\mathrm{value}:$ <source>$)\{\}$ where source and targets refers to some data type. For example $\mathrm{public\ NewPerson\ fromPerson(o:\ Person)\{\}}$ where the data type has changed from *Person* to *NewPerson*.

The proposed evolution process targets schema evolution. As such, only state which exists in the stateful dataflow graph is taken into account. This can be compared to the distinction between the set of DDL statements versus DML statements in SQL. The schema can be altered implying that the underlying state is made compatible, but no alterations that do not relate to schema compatibility are executed. With an x amount of physical state elements in the source schema (rows in relational databases), the target state will also contain x elements. In section 4.3, open-source projects are discussed which elaborate more on the difference. *Hydrating* state from external systems or files stored on a filesystem is not supported.

Renaming a state handle (S2 as defined in section 3.4) could be applied cross operator. On partitioned streams however, the state is often stored relative to the partition key (note that the key is stored as part of the state schema in Table 3.1). When moving state cross-operator where the source operator has a different key type with respect to the target operator, the state should be re-keyed. Non-partitioned state does not have this limitation. Based on these challenges, renaming state handles cross operator is left out of scope.

The state attribute extraction mechanism visualized in Figure 3.1 does not proxy to a runtime environment during the evolution phase. During evolution, these runtime attributes are not known. The *stop-and-restart* mechanism handles the state migration in an *offline* scenario such that the runtime context is not required. To use the schema with dynamic reconfiguration, e.g. using *pause-and-resume*, the extraction context should proxy to the runtime context. The extraction context should be cleared and re-calculated on each reconfiguration of the runtime context that the extraction context proxies.

The performed work does not feature migration of window state or iterative streams.

4

Evaluation

This chapter will elaborate on the evaluation of the work and conducted experiments. In section 4.1, the setup to evaluate correctness is described. In section 4.2, the NEXmark benchmark [87] is described and executed with hypothesized changes.

4.1. Evaluating the correctness of the evolution process

In order to validate correctness of the generated schemas, *synthetic* streaming jobs are created containing state. These streaming jobs attempt to capture all scenarios described in subsection 3.4.1 for various state types and state descriptors.

To generate events for the experiments, a custom source function is introduced. The custom source function uses the $\mathrm{TypeTag}$ feature of Scala to know the generic datatype used to instantiate the class at runtime. The source function then emits an event every 1 second with a random value using the Java Random class. To construct strings, 5 characters are drawn from a distribution containing only letters.

When the scenario relates to a keyed stream, a custom key selector is used which partitions the incoming stream into two output streams. For natural numbers, the stream is partitioned into even and odd. For real numbers, the stream is partitioned into > 0 or ≤ 0. Booleans are partitioned based on their truthiness. For strings, the sum of alphabetical indices is used to yield a natural number. The natural number is then again used to partition into odd or even.

The sink of the scenario stores all incoming values in a generic key-value store. The store is available statically while the results are queryable using the unique identifier of the sink. This allows the extraction of both the key-value pairs from the source dataflow graph as well as the target dataflow graph.

To simulate keyed state, a custom version of the RichMapFunction interface is used. The RichMapFunction is an extension of the AbstractRichFunction interface which provides access to the state.

- ValueState is created by persisting the latest event
- ListState is created by creating a buffer of length 5 and persisting the latest event at a random position
- MapState is created by persisting the latest event as both the key and the value
- ReducingState is created by retaining either the latest event or the previously retained value

To create operator state, a custom version of the MapFunction interface is used which implements the CheckpointedInterface. The operator tracks the latest value that is mapped. When the snapshotState lifecycle hook is called, the latest event is persisted. BroadcastState is created using a custom version of the BroadcastProcessFunction. BroadcastState uses the MapState data structure and thus relies on the same implementation as described above.

When creating window state, a custom reduce function implementing the ReduceFunction interface is used. On each reduce call, a random value of the two incoming values is selected to be kept.

4.2. NEXmark benchmark

The NEXmark benchmark features eight different real world streaming query scenarios as described in [87]:

- **Q1 —** Currency conversion
- **Q2 —** Selection
- **Q3 —** Local Item Suggestion
- **Q4 —** Average Price for a Category
- **Q5 —** Hot Items
- **Q6 —** Average Seller Price by Seller
- **Q7 —** Highest Bid
- **Q8 —** Monitor New Users

Query 1 is used to measure the redeployment time of a *real world* change when no state is involved. A hypothesized change is introduced to the query which then runs the evolution process. Query 4 is used to measure the deployment time when state migration occurs. Hypothesized changes are introduced which affect the keying of the event stream and the state related to categorizing bids.

4.2.1. Experimental Setup

Query 1 streams a bounded list of events. An event can either be a bid placed for some auction by some person, a new person entering an auction, or a new auction which has started. The source node in the dataflow graph generates these events based on a generator supplied by Apache Beam [62]. The source stream is connected to a flat map operation which filters all non-bid events. This yields a stream of solely bids. The bid stream is then mapped where each bid is converted using an artificial conversion rate to now contain a price in euros instead of dollars. The converted bids are finally sent to a sink for storage. The entire dataflow graph is stateless and can therefore be used to measure the impact of the *stop-and-restart* approach agnostic to the contained state. The result is visualized in Figure 4.1.

Figure 4.1: Query 1 ‘migration’

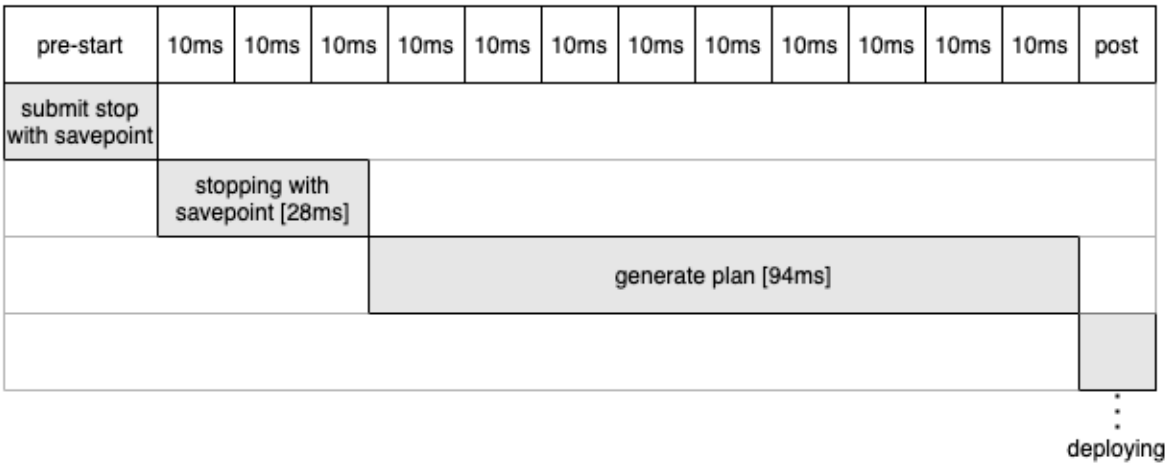


Table 4.1: Query 4 experiment parameters, state sizes and timings.

# Events	Resulting state size (MB's)	(re)-processed rows	SWS (ms)	GP (ms)	M (ms)
1.000	0,00556	62	18 ± 2.00	9 ± 3.63	2192 ± 21.47
1.000.000	0,942	60.010	91 ± 1.49	9 ± 2.72	2256 ± 17.35
10.000.000	9,15	600.010	176 ± 51.71	11 ± 2.16	4069.5 ± 60.21
25.000.000	22,8	1.500.010	217 ± 33.48	13 ± 3.63	7254.4 ± 515.83
50.000.000	45,7	3.000.010	273 ± 45.35	13.5 ± 5.19	11941 ± 1362.97
100.000.000	91,5	6.000.010	619 ± 91.14	27.5 ± 3.32	22693.5 ± 760.82

Query 4 is more complex and requires state. In a similar fashion as query 1, a source node is created. Two flat map operations are attached to the source stream. The first flat map filters all non bid events resulting in a stream of solely bids. The second flat map filters all non-auctions, resulting in a stream of solely auctions. The bid stream is keyed based on the unique identifier of the auction. The auction stream describes a broadcast state descriptor which will provide a lookup map for an auction identifier to a category. The broadcast stream is connected with the bid stream and a process operator is applied which will process incoming bids, use the broadcast state to derive a category for the bid and output a tuple of the category and price. The resulting tuple stream is keyed based on the category to maintain state solely for the categorical keys. This keyed stream is then mapped with a rich map operator which tracks the amount of bids for some specific category and the current average. For each incoming bid, the counter is increased and the average recalculated before being persisted as keyed state. The map operation returns a tuple of the category and the current average which is sent to a sink.

The experiment is repeated for a monotonically increasing number of events generated by the NEXmark generator. The parameters, sizes, and timings are reported in Table 4.1. A visualization is shown in Figure 4.2. SWS, GP, and M are respectively *stopping with savepoint*, *generate plan*, and *migrating*.

The results as shown in Table 4.1 are achieved using a Ryzen 7 3700X running under WSL2 in Windows 10. The state is persisted to an Intel 660p 1TB drive. The JVM is warmed up by running the test once without using the results. This is repeated ten times for each configuration yielding the mean and standard deviations as reported in Table 4.1. From Table 4.1, it can be seen that the stopping with a savepoint increases with the amount of events processed. Interestingly enough, generating a plan (GP) for evolution takes longer the more events are processed. The timings are reported in milliseconds, so minor changes in the availability of resources can highly impact the reported latencies. The reported timings show that there is a minimum latency of approximately 2 seconds when migrating state. This is the result of initializing the migration execution environment for the state processor API. The latency induced by reprocessing state scales approximately linear to the amount of rows processed (and resulting state size). The generate plan phase deviates slightly although this time is assumed to be stable (the dataflow graph does not change). The timings are reported in milliseconds, minor deviations in available system resources can impact this phase such as remaining garbage collection.

4.3. Github projects

Github [1] is used to find projects using the state processor API. The deployment mechanism, described in section 3.6, uses the state processor API to deploy the target application with updated state. Projects are selected based on the search term $\mathrm{BootstrapTransformation}$ or $\mathrm{OperatorTransformation}$. Results where the project is a *fork* of Apache Flink or *contains* the Apache Flink codebase are excluded from the table in Table 4.2.

The *vip-Augus/flink-learning-note* Github project, contains a custom transformation class named $\mathrm{OperatorTransformation}$. This is unrelated to the state processing API and excluded from the table. The *leonardBang/flink-sql-etl* Github project only contains a $\mathrm{SavepointMetadata}$ file referencing

[1] $\mathrm{https://github.com}$

Figure 4.2: Query 4 migration for 1.000.000 events

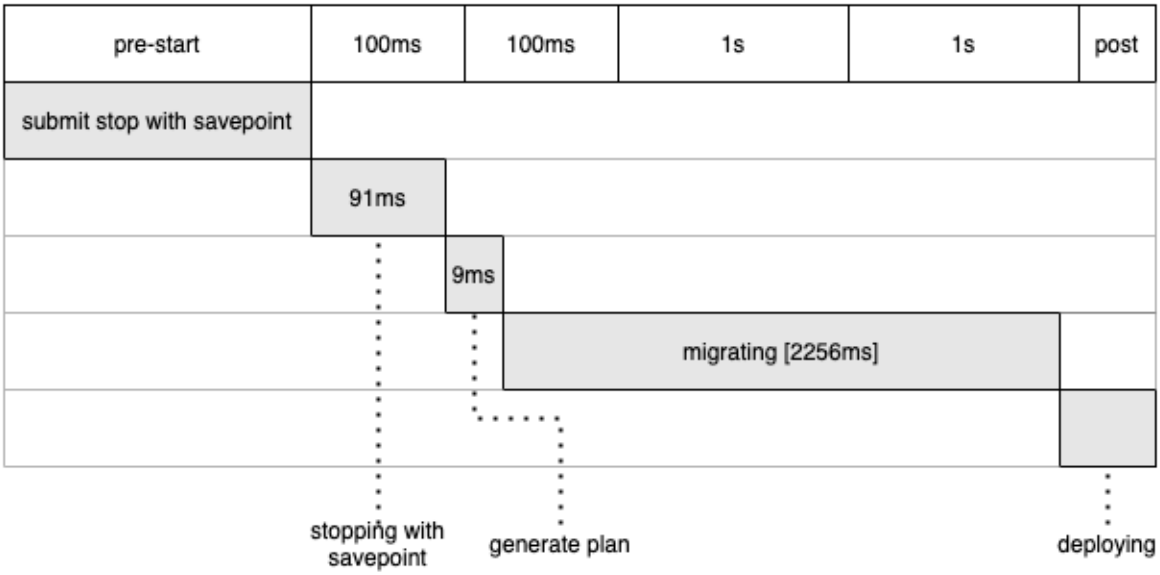


Table 4.2: Github projects using the State Processor API

Project Name	**Search Term** [2]	Schema	State
streamnative/pulsar-flink-state-migrate	BT,OT	depends	depends
ChenShuai1981/flink190	BT,OT	yes	no
minmay/flink-patterns	BT,OT	yes	no
hqbhoho/learn-bigdata	BT,OT	yes	no
alpinegizmo/timing-explorer	BT	no	no
segmentio/flink-state-management	OT	yes	no
wikimedia/wikidata-query-rdf	OT	yes	no
peterdeka/stateproc	OT	yes	no

bootstrap transformation. This file does not use the state processing API and is excluded from the table. In *cheegoday/flink_djg* and *mumudong/forfun*, a savepoint is written to and read from but no corresponding job is present. These projects are excluded from the table.

In Table 4.2 the *Schema* column represents the hypothesis that the schema can be extracted from the job. The *State* column represents whether the state can be migrated using the performed work described in chapter 3. Note that the Schema and State columns are the expected outcome by performing a static analysis on the Github project. These analysis are explained per project in subsection 4.3.1 and subsection 4.3.2.

Addition of an operator is an identified evolution scenario (see section 3.4). Addition of an operator can however be subdivided into hydrated addition and non-hydrated addition. This partitioning is important for the discussed Github projects as the performed work and schema migration assume pre-existing state. The *state processor API* can also be used to initialize an operator with not previously existing state. The projects are categorized into either one of these partitions and discussed separately.

4.3.1. Pre-existing state migrations

streamnative/pulsar-flink-state-migrate handles the migration of the $\mathrm{pulsar\text{-}flink\text{-}connector}$. The migration cannot be reproduced using the unary and binary schema evolution models as the new state handles depends on data from multiple sources combined using a cross join. When the *pulsar-flink-connector* $\mathrm{subName}$'s exist in the source, the dataflow graph evolution can be captured using:

```
val s = new TupleTypeInfo[Tuple2[String, MessageId]](
  of(classOf[String]), of(classOf[MessageId]))
val t = new TupleTypeInfo[Tuple2[TopicSubscription, MessageId]](
  of(classOf[TopicSubscription]), of(classOf[MessageId]))

StateMigration.operator()
  .setSource("uid", "topic-partition-offset-states")
  .setTarget("uid", "topic-offset-states")
  .map(s, t, (f: Tuple2<String, MessageId>) =>
     Tuple2.of(TopicSubscription.builder().topic(tuple2.f0).build(), tuple2.f1)
   )
```

In *alpinegizmo/timing-explorer*, the state backend is changed. Using the *same-code* approach to migrate the state backend, a replication of the current data stream is mimicked in the state processing API environment. All previously existing state is extracted from an existing savepoint and rewritten to a new savepoint of the updated state backend. While the evolution process of chapter 3 can extract and re-add the state and even derive all state handles implicitly (unchanged is a *special* type of unary evolution), the implemented version does not support changing state backends as state backends are of a unified format since Apache Flink 1.13.

In *segmentio/flink-state-management*, the schema remains unchanged but can be derived from the job. The state processor is used to filter previously existing state, this is a type of DML operation while the performed work targets DDL migrations.

4.3.2. Non existing state hydration

ChenShuai1981/flink190 implements a BootstrapStateProcessorApiDemo for self learning purposes. Instead of using an existing savepoint to extract state from, the state is defined inline as a list of values. Ignoring the hydrated state, the evolution can be captured explicitly using:

```
StateMigration.global()
  .setTarget("currency_converter")
  .addition()
StateMigration.global()
  .setTarget("summarize")
  .addition()
```

Note that the addition and removal of operators without previous state can already be resolved implicitly by an unmodified version of Apache Flink 1.12. A similar procedure is applied in *minmay/flink-patterns* where the state is extracted from an external system using JDBC. Ignoring the extracted state, the evolution can be captured using:

```
StateMigration.global()
  .setTarget("boot-strap")
  .addition()
```

Similarly, *peterdeka/stateproc*, *hqbhoho/learn-bigdata* and *wikimedia/wikidata-query-rdf* only hydrate state. In *peterdeka/stateproc*, a stream of serialized objects is read from a file as a tuple containing five elements. The tuples are hydrated to a new operator which maintains three state handles. In *wikimedia/wikidata-query-rdf*, the UpdaterBootstrapJob reads Apache Kafka offsets from a CSV file and hydrates the offsets to an operator.

5

Conclusion

To conclude the thesis, the established research questions, as defined in section 1.6, are answered.

- **RQ1 —** How can a stateful dataflow graph be transformed to a state schema?

In chapter 3, the outline of generating the stateful dataflow graph schemas is provided. An implementation is provided for a modified version of Apache Flink using static software artifacts. In section 3.3, the attributes of the stateful dataflow schema are listed which refer to dataflow graph properties such as partitioning, windowing and processing. The provided implementation relies on the user code exposing access to the native transformations which construct the stateful dataflow graph [1]. These transformations contain either statically the collected attributes or provide runtime access which are collected using an *extraction runtime environment*. The result is a state schema based representation of the stateful dataflow graph.

- **RQ2 —** Given a state schema for a stateful dataflow graph, what evolution scenarios exist?

The discretized evolution scenarios on the schema are defined in section 3.4. The scenarios are based on common evolutions in the relational database domain [51], research on dynamic reconfiguration [20], and empirical usage. The scenarios can be roughly categorized into stateful evolutions (S1-S4), operator evolutions (S5-S8), and partitioning evolutions (S9). The collected attributes that are part of the schema reflect the attributes required to explicitly define (or implicitly derive some of) these evolutions.

- **RQ3 —** Which transformations can be derived without user intervention that migrate state for stateful dataflow schema evolutions?

Using the schema, *diffing* can be applied on two instances of the schema, a process detailed in section 3.4. Based on a view of what has changed between two instances of a schema, mappings of primitive evolutions can be derived. The predefined set of evolution scenarios allows scoping of the possible set of analyses to perform on the two schema versions. The remaining question is then: what scenarios support implicit migration and what are their constraints? In subsection 3.4.1, for each scenario their implicit behavior is discussed. With the defined set of discretized evolutions, the unary evolutions are possible to implicitly derive (addition, removal) while binary migrations (update, move) require explicit definition. In section 6.3, a probability model for deriving implicit transformations is discussed.

- **RQ4 —** How can two instances of stateful dataflow graph schemas be used for (dynamic) reconfiguration?

[1] Note that these transformations refer to the construction of the DAG, not transformations on the schema.

The implementation described in section 3.5 demonstrates a *stop-and-restart* reconfiguration using the *code-to-schema* approach. The re-processing of existing state in a stateful dataflow graph with respect to Apache Flink is categorized as *same code*, described in subsection 2.5.2. Instead with a *code-to-schema* approach, a stateful schema view on the running instance of the dataflow graph can be used for extraction of existing state. The view on the target schema of the running job is available, where the state should be *loaded* into. The transformations that were defined using either explicit definition or implicit derivation represent the transforming of the state before loading. On a high-level, the *stop-and-restart* follows an *extraction-transform-load* (ETL) process.

Advantages of the *code-to-schema* approach opposed to the *same-code* include expressiveness and fault tolerance:

- **expressiveness —** as only evolution scenarios have to be described. Implicitly derivable evolution scenarios can even be left out. In the *same-code* model, to hydrate state, all state should be explicitly defined. To define evolution scenarios, only the conversion of data types uses custom user defined implementations. The definition is agnostic to the re-processing model used as opposed to the *same-code* which currently explicitly targets *stop-and-restart* reprocessing.
- **fault tolerance —** as the generated schema captures and validates evolution scenarios. When evolving the state schema using the *same-code* approach, a human-error in the definitions of state descriptors can go unnoticed as no validation.

Disadvantages of the *code-to-schema* approach opposed to the *same-code* include flexibility:

- **flexibility —** as the *same-code* approach directly uses the state processor API with complete manual control over the re-processing. For example, hydration of not pre-existing state. When operators are removed or added, the re-processing can modify an existing savepoint instead of creating one. The *code-to-schema* approach (or the underlying libraries) can be improved to support these operations, see section 6.3.

The problem statement in section 1.4, illustrated the problem of (dynamic) reconfiguration and the research topics this thesis covered. With the high load, i.e. 50 million deployments in 12 months [89], the need for a runtime of SFaaS applications to support evolutions becomes clear. The performed work hopes to contribute to the road of being able to use distributed stream processors as a runtime with a higher fault tolerance and expressive evolution syntax.

6

Discussion

In this chapter, a discussion on the implemented techniques is provided. This discussion will bridge some of the discussed background material and related work with the process proposed in chapter 3. The proposed pipeline can be viewed as a proof of concept on stateful dataflow graph evolution, engineering challenges not related to the execution of the proposed job evolution pipeline were left out and will be discussed in this chapter.

6.1. Comparison to graph based schema evolution

With a stateful dataflow graph represented as a DAG, it seems like a logical step to resolve differences using graph theory. This concept is used in Terraform [83] where infrastructure for public cloud providers is modeled as a DAG. In Terraform, the future *state* of the infrastructure is referenced as '*Config*' while the current state is simply referred to as '*State*'. The state is described using HCL [40], a configuration format for blocks of (hierarchical) key-value pairs. A graph can be constructed by defining relations between blocks. On the node level, the attributes of a node are compared on a key-value basis and reduced to change primitives. Similarly, the HyperGraph model [56] models a state schema of a relational database where edges represent inter-node relations. The nodes in distributed stateful dataflow graphs do not contain relations to other nodes. The DAG model is used to describe the passing of events, not the management of state. Without relations between nodes, the state schema can be modeled as a set instead of a graph.

6.2. Contributions

The main contributions of this thesis can be summarized into two parts. First, a design and implementation are shown that derive state schemas from stateful dataflow graph definitions. Second, the state schema definitions are used to determine incompatible state and reprocess state in case of evolutions. As the schema generation steers the state migration, the evolution process can be referred to as *code-to-schema*.

Apache Flink provides the business logic to create and run stateful dataflow graphs. The Apache Flink State Processor API provides low-level manual control over the state contained in stateful dataflow graphs by manipulating or hydrating operators. For the stateful dataflow graph process, an abstraction is created on top of the Apache Flink dataflow graph definition which exposes the execution environment and explicit migrations. This is what enables the *code-to-schema* approach as the schema is directly read from the dataflow graph definition. The remainder of the process including the REST endpoints are part of the newly developed *flink-migrate* package part of the modified Apache Flink codebase. The package is tested through the created *flink-end-to-end-tests/flink-migrate-test* package. The tests represent evolution scenarios.

The work performed by Ottenwälder et al. [65], developing a *Migration Plan* to migrate stateful operators, resembles a similar strategy towards schema-like migrations. Their implementation dif-

fers in that the *migration plan* describes the future states of operators without a global *state schema*. The *migration plan* assumes immutable data where the *code-to-schema* evolution is designed to also reprocess data.

The work of Bartnik et al. [20] describes dynamic reconfiguration using the *pause-and-resume* deployment approach. The discussion mentions that the approach has as an advantage over *stop-and-restart* that in-flight events during cancellation cannot be guaranteed to be exactly once processed. This behavior has changed with the introduction of *stop with savepoint* by Apache Flink [1]. This means that both approaches can guarantee exactly once processing semantics. The work uses the concept of *modification markers* which are injected into the data stream containing the modifications of the dataflow graph. No explanation other than that the modification marker is constructed in a *Modification Coordinator* is provided. Additionally, the state migration does not reprocess existing state. If the UDF contains state handles not backwards compatible with the previous UDF, the state migration fails. In the *code-to-schema* approach, this is handled through transformations attached to the evolution scenario.

6.3. Future improvements for Apache Flink

When sending a TRGR to invoke a job evolution process, the performed work asserts a trust on the supplied arguments. The software artifact used to start J_1 should instead be derived from the running instance. The target software artifact should also be uploaded to a location accessible by the stream processor. From the perspective of the stream processor, there is no versioning scheme in place that handles evolutions. An evolution takes a source and evolves to a target. Both the storage and versioning can be resolved using registries.

Versioning and distribution of software artifacts is often handled through registries. Common traits of registries are a namespace, identifier, and versioning scheme of the uploaded artifact. The DockerHub image registry has already been discussed. Other notable package managers are the Node Package Manager (NPM), The Python Package Index (PyPi), Maven Central, and Advanced Packaging Tool (APT).

A runtime using such a registry can track the artifact used to deploy a running instance of the job. The Amazon Web Services (AWS) platform provides the Elastic Container Service (ECS) [1] which bridges the gap between stored artifacts and a runtime using Task Definitions [4]. A Task Definition contains references to required software artifacts, infrastructural components, and runtime parameters.

Two new additions to Apache Flink would allow a similar process in Apache Flink. A registry and a service executor. The registry is where developers would upload packaged applications with a version identifier and the service executor manages the jobs lifecycle. The service executor loads a service definition describing execution semantics of the service to run such as the software artifact and its version. Updating the service definition allows for a new deployment which is managed by the service executor. A service executor acts as a gateway for a single service. The service executor can provide the locking mechanism required to block parallel evolutions of a service. The locking mechanism restricts additional evolution processes to be invoked while a job is under evolution. Only after the lock has been granted should the evolution process continue.

Building on the *code-to-schema* approach towards reconfiguration of stateful dataflow graphs, a deployment mechanism which allows for *pause-and-resume* or *dataflow replication* should be investigated. The schema approach is agnostic to the deployment model. Adopting the mechanism described by Bartnik et al. [20] could produce dynamic reconfiguration of the dataflow graph.

The migrations are applied *all-at-once*. Although somewhat overlapping with the future work relating to *pause-and-resume* style migrations, the schema should be able to apply a list of scenarios as individual transformations to a running query. While this would simplify the burden on the migration process, it would also add additional challenges such as providing *down* type migrations to rollback to a previous schema.

While the implicit migrations can be detected based on schema differences, explicit transformations offer more predictable behavior. The predicate based migration scenarios of the performed

[1] https://aws.amazon.com/ecs/

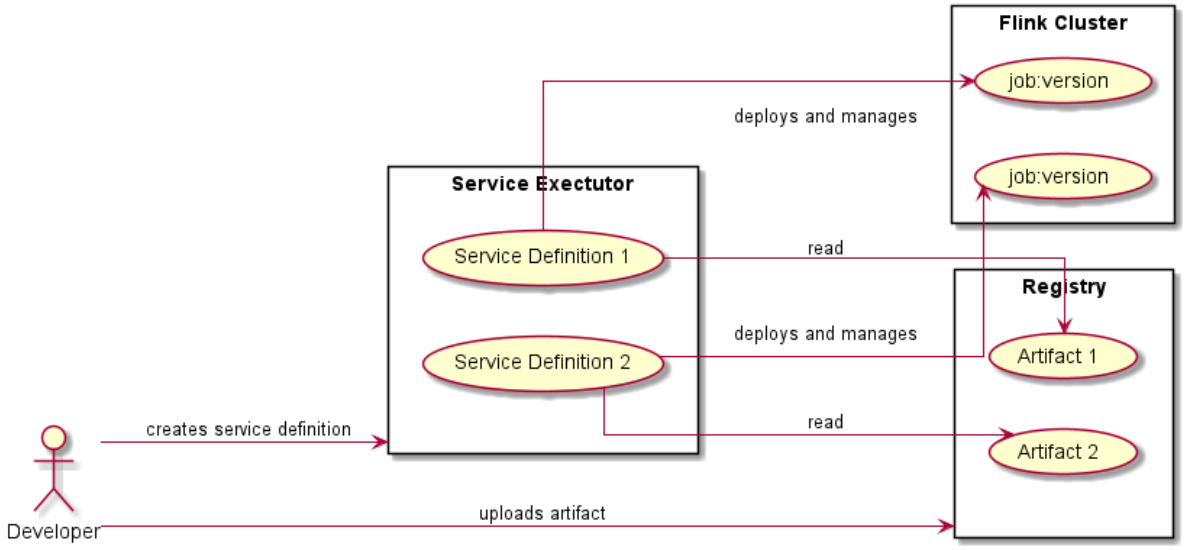


Figure 6.1: Example Apache Flink architecture with a registry

work contain all information to generate source code for explicit migrations which can be then supplied by the developer. With this implicit to explicit behavior, a probability model of deriving other migrations can be possible. The user would confirm or reject implicit migrations from the probability model.

When the *time-to-live* (TTL) of state is configured, state can be automatically discarded after some period of time. Taking TTL into account when performing state migrations can yield interesting migration scenarios where the old job can be put in a draining state until all state has expired (while the new state is handling new events that would not update the draining state).

The evolution process (chapter 3, *code-to-schema*), is used for DDL based schema evolutions. This assumes pre-existing state. Similarly, to the relational database domain, a schema approach can be used for DML and DQL like operations.

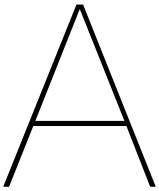
- **DML —** can be subdivided into DML as part of the schema evolution process (hydrating) or DML operations as part of a running dataflow graph. Hydration is discussed as part of the evaluation, see section 4.3. Dynamic DML can rely on the schema to generate modification markers (similar to [20]). The implementation of such a schema based DML approach can follow ORM like approaches:

  $\mathrm{StateMigration.keyed()}$
  $\mathrm{.setTarget(”some\text{-}uid”,\ ”some\text{-}state\text{-}key”)}$
  $\mathrm{.update(UpdateFunction)}$
  $\mathrm{.where(Predicate)}$

  The placeholders ($\mathrm{UpdateFunction}$ and $\mathrm{Predicate}$) should be replaced with actual implementations.

- **DQL —** based on a schema can be applied on the running stateful dataflow graph. For state consistency, a mechanism similar to the checkpointing and savepointing mechanisms can be used to execute DQL statements. Injecting a barrier in the event stream, propagated to all operators as opposed to mechanisms which enable direct access to operators [69]. This ensures the same state consistency level as the native stateful dataflow graph. The barrier contains a *query scenario* based on the schema and is decorated with the respective state on each operator it passes.

Appendices



Common Evolution Scenarios

The following list is extracted from Zero-Downtime SQL Database Schema Evolution for Continuous Deployment [51]:

- **S1:** Adding a non-nullable column to an existing table
- **S2:** Adding a nullable column to an existing table
- **S3:** Renaming a non-nullable column
- **S4:** Renaming a nullable column
- **S5:** Dropping a non-nullable column
- **S6:** Dropping a nullable column
- **S7:** Modifying the data type of a non-nullable column
- **S8:** Modifying the data type of a nullable column
- **S9:** Modifying the data type of a non-nullable column from integer to text
- **S10:** Making a non-nullable column nullable
- **S11:** Making a nullable column non nullable
- **S12:** Modifying the default value of a non-nullable column
- **S13:** Modifying the the default value of a nullable column
- **S14:** Creating a foreign key constraint on a non-nullable column
- **S15:** Creating a foreign key constraint on a nullable column
- **S16:** Creating an index on an existing non-nullable column
- **S17:** Renaming an existing index
- **S18:** Dropping an existing index
- **S19:** Renaming an existing table

Bibliography

[1] *(DEPRECATED) Apache Flink User Mailing List archive. - Stop vs Cancel with savepoint*. URL: http://apache-flink-user-mailing-list-archive.2336050.n4.nabble.com/Stop-vs-Cancel-with-savepoint-td41865.html.

[2] Adil Akhter and Marios Fragkoulis. *Deploying Stateful FaaS on Streaming Dataflows - Adil Akhter & Marios Fragkoulis - YouTube*. Oct. 2019. URL: https://www.youtube.com/watch?v=wKfzDPkbAao&t=337s.

[3] Adil Akhter, Marios Fragkoulis, and Asterios Katsifodimos. "Stateful functions as a service in action". In: *Proceedings of the VLDB Endowment*. Vol. 12. 12. VLDB Endowment, Aug. 2018, pp. 1890–1893. DOI: 10.14778/3352063.3352092. URL: https://dl.acm.org/doi/10.14778/3352063.3352092.

[4] *Amazon ECS task definitions - Amazon Elastic Container Service*. URL: https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_definitions.html.

[5] *Apache Avro 1.10.2 Specification*. URL: http://avro.apache.org/docs/current/spec.html#Schema+Resolution.

[6] *Apache Flink 1.12 Documentation: Data Types & Serialization*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/types_serialization.html.

[7] *Apache Flink 1.12 Documentation: Fault Tolerance via State Snapshots*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.12/learn-flink/fault_tolerance.html.

[8] *Apache Flink 1.12 Documentation: Glossary*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.12/concepts/glossary.html#flink-cluster.

[9] *Apache Flink 1.12 Documentation: State Backends*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.12/ops/state/state_backends.html#state-backends.

[10] *Apache Flink 1.12 Documentation: State Schema Evolution*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/stream/state/schema_evolution.html.

[11] *Apache Flink 1.12 Documentation: Upgrading Applications and Flink Versions*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.12/ops/upgrading.html#application-state-compatibility.

[12] *Apache Flink 1.12 Documentation: Working with State*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/stream/state/state.html.

[13] *Apache Flink: A Deep Dive into Rescalable State in Apache Flink*. URL: https://flink.apache.org/features/2017/07/04/flink-rescalable-state.html.

[14] *Apache Flink: What is Apache Flink? Architecture*. URL: https://flink.apache.org/flink-architecture.html.

[15] *Apache Heron ů A realtime, distributed, fault-tolerant stream processing engine*. URL: https://heron.incubator.apache.org/.

[16] *AWS Lambda Serverless Compute - Amazon Web Services*. URL: https://aws.amazon.com/lambda/.

[17] *AWS Step Functions | Serverless Microservice Orchestration | Amazon Web Services*. URL: https://aws.amazon.com/step-functions.

[18] Ioana Baldini et al. "Serverless Computing: Current Trends and Open Problems". In: *Research Advances in Cloud Computing*. Ed. by Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya. Singapore: Springer Singapore, 2017, pp. 1–20. ISBN: 978-981-10-5026-8. DOI: 10.1007/978-981-10-5026-8{_}1. URL: https://doi.org/10.1007/978-981-10-5026-8_1.

[19] Jay Banerjee et al. “Semantics and Implementation of Schema Evolution in Object-Oriented Databases”. In: *ACM SIGMOD Record* 16.3 (Dec. 1987), pp. 311–322. ISSN: 01635808. DOI: 10.1145/38714.38748. URL: https://dl.acm.org/doi/10.1145/38714.38748.

[20] Adrian Bartnik et al. “On-the-fly Reconfiguration of Query Plans for Stateful Stream Processing Engines”. In: *BTW 2019* (2019). Ed. by Torsten Grust et al., pp. 127–146. DOI: 10.18420/btw2019-09.

[21] Philip A Bernstein and Sergey Melnik. “Model Management 2.0: Manipulating Richer Mappings”. In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’07. New York, NY, USA: Association for Computing Machinery, 2007, pp. 1–12. ISBN: 9781595936868. DOI: 10.1145/1247480.1247482. URL: https://doi.org/10.1145/1247480.1247482.

[22] *Build your Python image | Docker Documentation*. URL: https://docs.docker.com/language/python/build-images/.

[23] L Burkholder. “The halting problem”. In: *ACM SIGACT News* 18.3 (Apr. 1987), pp. 48–60. ISSN: 0163-5700. DOI: 10.1145/24658.24665. URL: https://dl.acm.org/doi/10.1145/24658.24665.

[24] Paris Carbone et al. “Beyond Analytics: The Evolution of Stream Processing Systems”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Vol. 8. 20. New York, NY, USA: Association for Computing Machinery, June 2020, pp. 2651–2658. ISBN: 9781450367356. DOI: 10.1145/3318464.3383131. URL: https://dl.acm.org/doi/10.1145/3318464.3383131.

[25] Paris Carboney et al. “State management in Apache Flink: ő consistent stateful distributed stream processing”. In: *Proceedings of the VLDB Endowment*. Vol. 10. 12. Association for Computing Machinery, Aug. 2017, pp. 1718–1729. DOI: 10.14778/3137765.3137777. URL: https://dl.acm.org/doi/10.14778/3137765.3137777.

[26] Guoqiang Jerry Chen et al. “Realtime Data Processing at Facebook”. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD ’16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1087–1098. ISBN: 9781450335317. DOI: 10.1145/2882903.2904441. URL: https://doi.org/10.1145/2882903.2904441.

[27] *curl*. URL: https://curl.se/.

[28] Miyuru Dayarathna and Srinath Perera. “Recent Advancements in Event Processing”. In: *ACM Comput. Surv.* 51.2 (Feb. 2018). ISSN: 0360-0300. DOI: 10.1145/3170432. URL: https://doi.org/10.1145/3170432.

[29] D J De Witt. “Direct A Multiprocessor Organization for Supporting Relational Database Management Systems”. In: *IEEE Trans. Comput.* 28.6 (June 1979), pp. 395–406. ISSN: 0018-9340. DOI: 10.1109/TC.1979.1675379. URL: https://doi.org/10.1109/TC.1979.1675379.

[30] *Deployment Strategies - Introduction to DevOps on AWS*. URL: https://docs.aws.amazon.com/whitepapers/latest/introduction-devops-aws/deployment-strategies.html.

[31] Jianbing Ding et al. “Optimal Operator State Migration for Elastic Data Stream Processing”. In: (Jan. 2015). URL: https://arxiv.org/abs/1501.03619v5.

[32] Dominik Ernst, Alexander Becker, and Stefan Tai. “Rapid Canary Assessment Through Proxying and Two-Stage Load Balancing”. In: *Proceedings - 2019 IEEE International Conference on Software Architecture - Companion, ICSA-C 2019* (May 2019), pp. 116–122. DOI: 10.1109/ICSA-C.2019.00028.

[33] Yi Hsuan Feng, Nen Fu Huang, and Yen Min Wu. “Efficient and adaptive stateful replication for stream processing engines in high-availability cluster”. In: *IEEE Transactions on Parallel and Distributed Systems* 22.11 (2011), pp. 1788–1796. DOI: 10.1109/TPDS.2011.83.

[34] Marcelo Fiore and Marco Devesas Campos. “The Algebra of Directed Acyclic Graphs”. In: *Computation, Logic, Games, and Quantum Foundations. The Many Facets of Samson Abramsky: Essays Dedicated to Samson Abramsky on the Occasion of His 60th Birthday*. Ed. by Bob Coecke, Luke Ong, and Prakash Panangaden. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 37–51. ISBN: 978-3-642-38164-5. DOI: 10.1007/978-3-642-38164-5{_}4. URL: https://doi.org/10.1007/978-3-642-38164-5_4.

[35] Avrilia Floratou et al. “Dhalion: Self-Regulating Stream Processing in Heron”. In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1825–1836. ISSN: 2150-8097. DOI: 10.14778/3137765.3137786. URL: https://doi.org/10.14778/3137765.3137786.

[36] Martin Fowler. *BlueGreenDeployment*. 2010. URL: https://martinfowler.com/bliki/BlueGreenDeployment.html.

[37] Pedro García López et al. “Comparison of FaaS Orchestration Systems”. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 2018, pp. 148–153. DOI: 10.1109/UCC-Companion.2018.00049.

[38] *GitHub - facebook/rocksdb: A library that provides an embeddable, persistent key-value store for fast storage*. URL: https://github.com/facebook/rocksdb.

[39] *GitHub - flyway/flyway: Flyway by Redgate Database Migrations Made Easy.* URL: https://github.com/flyway/flyway.

[40] *GitHub - hashicorp/hcl: HCL is the HashiCorp configuration language.* URL: https://github.com/hashicorp/hcl.

[41] *GitHub - liquibase/liquibase: Main Liquibase Source*. URL: https://github.com/liquibase/liquibase.

[42] *GitHub - qubole/spark-state-store: Rocksdb state storage implementation for Structured Streaming*. URL: https://github.com/qubole/spark-state-store.

[43] Scott Hendrickson et al. “Serverless Computation with OpenLambda”. In: *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*. HotCloud’16. USA: USENIX Association, 2016, pp. 33–39.

[44] Martijn de Heus, Marios Fragkoulis, and Asterios Katsifodimos. “Distributed Transactions on Serverless Stateful Functions using Coordinator Functions”. PhD thesis. Delft University of Technology, 2021. URL: https://repository.tudelft.nl/islandora/object/uuid%3A25b6e54a-116a-444f-9cb7-693d595bb058.

[45] Moritz Hoffmann Andrea Lattuada Frank McSherry Vasiliki Kalavri John Liagouris Timothy Roscoe et al. “Megaphone: Latency-conscious state migration for distributed streaming dataflows”. In: *Proceedings of the VLDB Endowment* 12.9 (2019). DOI: 10.3929/ethz-b-000387642. URL: http://doi.org/10.14778/3329772.3329777.

[46] Fabian Hueske and Vasiliki Kalavri. *Stream processing with Apache Flink: fundamentals, implementation, and operation of streaming applications*. First rele. O’Reilly Media, 2019. ISBN: 9781491974292.

[47] J Humble, C Read, and D North. “The deployment production line”. In: *AGILE 2006 (AGILE’06)*. 2006, 6 pp.–118. DOI: 10.1109/AGILE.2006.53.

[48] *Implementing State Management - Hortonworks Data Platform*. URL: https://docs.cloudera.com/HDPDocuments/HDP2/HDP-2.6.2/bk_storm-component-guide/content/storm-state-mgmt.html.

[49] Christopher Ireland et al. “A Classification of Object-Relational Impedance Mismatch”. In: *2009 First International Confernce on Advances in Databases, Knowledge, and Data Applications*. 2009, pp. 36–43. DOI: 10.1109/DBKDA.2009.11.

[50] Eric Jonas et al. “Cloud Programming Simplified: A Berkeley View on Serverless Computing”. In: *arXiv* (Feb. 2019). URL: http://arxiv.org/abs/1902.03383.

[51] Michael de Jong, Arie van Deursen, and Anthony Cleve. “Zero-Downtime SQL Database Schema Evolution for Continuous Deployment”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 2017, pp. 143–152. DOI: 10.1109/ICSE-SEIP.2017.5.

[52] *Kafka Streams Internal Data Management - Apache Kafka - Apache Software Foundation*. URL: https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Internal+Data+Management.

[53] A ; Katsifodimos and M Fragkoulis. “Operational stream processing: Towards scalable and consistent event-driven applications”. In: 2019 (2019), pp. 682–685. DOI: 10.5441/002/edbt.2019.86. URL: https://doi.org/10.5441/002/edbt.2019.86.

[54] Mariam Kiran et al. “Lambda architecture for cost-effective batch and speed big data processing”. In: *2015 IEEE International Conference on Big Data (Big Data)*. Institute of Electrical and Electronics Engineers Inc., Dec. 2015, pp. 2785–2792. DOI: 10.1109/BIGDATA.2015.7364082.

[55] Luo Mai et al. “Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems”. In: *Proc. VLDB Endow.* 11.10 (June 2018), pp. 1303–1316. ISSN: 2150-8097. DOI: 10.14778/3231751.3231765. URL: https://doi.org/10.14778/3231751.3231765.

[56] Peter McBrien and Alexandra Poulovassilis. “Schema Evolution in Heterogeneous Database Architectures, A Schema Transformation Approach”. In: *Proceedings of the 14th International Conference on Advanced Information Systems Engineering*. CAiSE ’02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 484–499. ISBN: 354043738X.

[57] Alok Mehta, David L Spooner, and Martin Hardwick. *Resolution of Type Mismatches in an Engineering Persistent Object System*. Tech. rep. Computer Science Dept., Rensselaer Polytechnic Institute, 1993.

[58] Marcelo R.N. Mendes, Pedro Bizarro, and Paulo Marques. “A Performance Study of Event Processing Systems”. In: *Performance Evaluation and Benchmarking*. Ed. by Raghunath Nambiar and Meikel Poess. Vol. 5895 LNCS. Springer Berlin Heidelberg, 2009, pp. 221–236. ISBN: 978-3-642-10424-4. URL: https://link.springer.com/chapter/10.1007/978-3-642-10424-4_16.

[59] Léonard Michel, José Andany, and Carole Palisser. “Management Of Schema Evolution In Databases”. In: *17th International Conference on Very Large Data Bases*. Ed. by Guy M. Lohman, Amilcar Sernadas, and Rafael Camps. Morgan Kaufmann, 1991, pp. 161–170. URL: https://www.researchgate.net/publication/221310520.

[60] *Netflix Innovator*. URL: https://aws.amazon.com/solutions/case-studies/netflix/.

[61] Sam Newman. *Monolith to Microservices: Evolutionary Patterns to Transform your Monolith*. 1st Editio. O’Reilly Media, Inc, USA, 2019. ISBN: 9781492047841.

[62] *Nexmark benchmark suite*. URL: https://beam.apache.org/documentation/sdks/java/testing/nexmark/.

[63] *NGINX Announces Results of 2016 App Dev & Delivery Survey*. URL: https://www.nginx.com/press/nginx-announces-results-of-2016-future-of-application-development-and-delivery-survey/.

[64] Helena Holmström Olsson, Hiva Alahyari, and Jan Bosch. “Climbing the "Stairway to Heaven" – A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software”. In: *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. 2012, pp. 392–399. DOI: 10.1109/SEAA.2012.54.

[65] Beate Ottenwälder et al. “MigCEP: Operator Migration for Mobility Driven Distributed Complex Event Processing”. In: *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*. DEBS ’13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 183–194. ISBN: 9781450317580. DOI: 10.1145/2488222.2488265. URL: https://doi.org/10.1145/2488222.2488265.

[66] *Peeking Behind the Curtains of Serverless Platforms | USENIX*. URL: https://www.usenix.org/conference/atc18/presentation/wang-liang.

[67] P Pietzuch et al. "Network-Aware Operator Placement for Stream-Processing Systems". In: *22nd International Conference on Data Engineering (ICDE'06)*. 2006, p. 49. DOI: 10.1109/ICDE.2006.105.

[68] *PostgreSQL: Documentation: 8.0: ALTER TABLE*. URL: https://www.postgresql.org/docs/8.0/sql-altertable.html.

[69] *Queryable State | Apache Flink*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/dev/datastream/fault-tolerance/queryable_state/.

[70] Young-Gook Ra. "Relational Schema Evolution for Program Independency". In: *Intelligent Information Technology*. Ed. by Gautam Das and Ved Prakash Gulati. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 273–281. ISBN: 978-3-540-30561-3.

[71] R Ramakrsihnan et al. "SRQL: Sorted Relational Query Language". In: *Proceedings. Tenth International Conference on Scientific and Statistical Database Management (Cat. No.98TB100243)*. 1998, pp. 84–95. DOI: 10.1109/SSDM.1998.688114.

[72] *RocksDB | A persistent key-value store | RocksDB*. URL: https://rocksdb.org/.

[73] Mohammad Sadoghi et al. "Efficient event processing through reconfigurable hardware for algorithmic trading". In: *Proceedings of the VLDB Endowment* 3.2 (Sept. 2010), pp. 1525–1528. ISSN: 21508097. DOI: 10.14778/1920841.1921029. URL: https://dl.acm.org/doi/10.14778/1920841.1921029.

[74] *Samza - State Management*. URL: http://samza.incubator.apache.org/learn/documentation/0.7.0/container/state-management.html.

[75] Tony Savor et al. "Continuous Deployment at Facebook and OANDA". In: *Proceedings of the 38th International Conference on Software Engineering Companion*. Austin, Texas: Association for Computing Machinery, 2016, pp. 21–30. ISBN: 9781450342056. DOI: 10.1145/2889160.2889223. URL: http://dx.doi.org/10.1145/2889160.2889223.

[76] Robert R. Schaller. "Moore's law: past, present, and future". In: *IEEE Spectrum* 34.6 (June 1997), pp. 52–59. DOI: 10.1109/6.591665.

[77] *Schema Evolution and Compatibility Confluent Documentation*. URL: https://docs.confluent.io/platform/current/schema-registry/avro.html#compatibility-types.

[78] Ben Shneiderman and Glenn Thomas. "An Architecture for Automatic Relational Database Sytem Conversion". In: *ACM Transactions on Database Systems* 7.2 (June 1982), pp. 235–257. ISSN: 0362-5915. DOI: 10.1145/319702.319724. URL: https://doi.org/10.1145/319702.319724.

[79] Andrea H Skarra and Stanley B Zdonik. "The Management of Changing Types in an Object-Oriented Database". In: *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA '86. New York, NY, USA: Association for Computing Machinery, 1986, pp. 483–495. ISBN: 0897912047. DOI: 10.1145/28697.28747. URL: https://doi.org/10.1145/28697.28747.

[80] *Spark Streaming - Spark 3.1.2 Documentation*. URL: https://spark.apache.org/docs/latest/streaming-programming-guide.html#checkpointing.

[81] *Stack Overflow Developer Survey 2020*. URL: https://insights.stackoverflow.com/survey/2020#technology-databases.

[82] *State Processor API | Apache Flink*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/libs/state_processor_api/.

[83] *Terraform by HashiCorp*. URL: https://www.terraform.io/.

[84] Quoc-Cuong To, Juan Soto, and Volker Markl. "A Survey of State Management in Big Data Processing Systems". In: *The VLDB Journal* 27.6 (Dec. 2018), pp. 847–872. ISSN: 1066-8888. DOI: 10.1007/s00778-018-0514-9. URL: https://doi.org/10.1007/s00778-018-0514-9.

[85] Edith Tom, Aybüke Aurum, and Richard Vidgen. “An exploration of technical debt”. In: *Journal of Systems and Software* 86.6 (2013), pp. 1498–1516. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2012.12.052. URL: https://www.sciencedirect.com/science/article/pii/S0164121213000022.

[86] Alexandre Torres et al. “Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design”. In: *Information and Software Technology* 82 (Feb. 2017), pp. 1–18. ISSN: 0950-5849. DOI: 10.1016/J.INFSOF.2016.09.009.

[87] Peter A. Tucker et al. “NEXMark A Benchmark for Queries over Data Streams DRAFT”. In: (2002). URL: https://datalab.cs.pdx.edu/niagaraST/NEXMark/.

[88] Giselle Van Dongen and Dirk Van Den Poel. “Evaluation of Stream Processing Frameworks”. In: *IEEE Transactions on Parallel and Distributed Systems* 31.8 (Aug. 2020), pp. 1845–1858. DOI: 10.1109/TPDS.2020.2978480.

[89] Werner Vogels. *The Story of Apollo - Amazons Deployment Engine - All Things Distributed*. 2014. URL: https://www.allthingsdistributed.com/2014/11/apollo-amazon-deployment-engine.html.

[90] Deepak Vohra. “Apache Avro”. In: *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*. Berkeley, CA: Apress, 2016, pp. 303–323. ISBN: 978-1-4842-2199-0. DOI: 10.1007/978-1-4842-2199-0{_}7. URL: https://doi.org/10.1007/978-1-4842-2199-0_7.

[91] *Wget - GNU Project - Free Software Foundation*. URL: https://www.gnu.org/software/wget/.

[92] Yali Zhu, Elke A Rundensteiner, and George T Heineman. “Dynamic Plan Migration for Continuous Queries over Data Streams”. In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’04. New York, NY, USA: Association for Computing Machinery, 2004, pp. 431–442. ISBN: 1581138598. DOI: 10.1145/1007568.1007617. URL: https://doi.org/10.1145/1007568.1007617.

[93] Igor Zinkovsky and Nikolay Topilski. *A Look at Software Performance Since 1940s*.