



# 스트림 처리 시스템의 진화에 관한 조사

Marios Fragkoulis<sup>1</sup> · Paris Carbone<sup>3,4</sup> · Vasiliki Kalavri<sup>5</sup> · Asterios Katsifodimos<sup>2</sup>

접수일: 2022년 8월 10일 / 수정일: 2023년 9월 1일 / 승인일: 2023년 9월 11일 / 온라인 게시: 2023년 11월 22일 © The Author(s) 2023

## Abstract

Stream 처리는 20년 이상 활발한 연구 분야였지만 최근 연구 커뮤니티와 전 세계 수많은 오픈 소스 커뮤니티의 성공적인 노력으로 인해 전성기를 맞이하고 있습니다. 이 설문 조사는 스트림 처리 시스템의 기본 측면과 비순차적 데이터 관리, 상태 관리, 내결합성, 고가용성, 로드 관리, 탄력성 및 재구성과 같은 기능 영역에서의 진화에 대한 포괄적인 개요를 제공합니다. 우리는 주목할 만한 과거 연구 결과를 검토하고, 1세대('00~'10)와 2세대('11~'23) 스트림 처리 시스템 간의 유사점과 차이점을 간략하게 설명하고, 미래 추세와 열려 있는 문제에 대해 논의합니다.

키워드 스트림 처리 · 내결합성 · 스트리밍 분석 · 클라우드 애플리케이션

## 1. 소개

스트림 처리 기술의 적용은 다시 부활하여 여러 가지 매우 다양한 산업에 침투했습니다. 오늘날 거의 모든 클라우드 공급업체는 관리형 스트림 처리 파이프라인 배포에 대한 최고 수준의 지원을 제공하는 반면, 스트리밍 시스템은 기존 스트리밍 분석(창, 집계, 조인 등)을 뛰어넘는 다양한 사용 사례에 사용됩니다. 몇 가지 예로는 동적 자동차 여행 가격 책정, 신용 카드 사기 감지, 예측 분석, 모니터링 및 실시간 교통 제어가 있습니다. 이 글을 쓰는 순간, 우리는 보다 일반적인 이벤트 중심 아키텍처 [96], 대규모 연속 ETL 및 분석, 마이크로 서비스 [91]를 구축하기 위해 스트림 프로세서를 사용하는 추세를 목격하고 있습니다.

지난 20년 동안 스트리밍 기술은 데이터베이스와 분산 시스템의 영향을 받아 크게 발전했습니다. 스트리밍 쿼리의 개념은 1992년 Tapestry 시스템에 의해 처음 소개되었으며 [148], 2000년대 초반에 스트림 처리에 대한 많은 연구가 이어졌습니다.

기본 개념과 아이디어는 데이터베이스 커뮤니티에서 시작되었으며 TelegraphCQ [48], Stanford의 STREAM, NiagaraCQ [51], Aurora/Borealis [12] 및 Gigascope [54]와 같은 프로토타입 시스템에서 구현되었습니다. 이러한 프로토타입은 데이터 모델에서는 대략적으로 동의했지만 의미론 쿼리에서는 상당히 달랐습니다 [21, 33]. 이 연구 기간에는 슬라이딩 윈도우 집계 [22, 107], 내결합성 및 고가용성 [30, 137], 로드 밸런싱 및 차단 [144]과 같은 다양한 시스템 문제도 도입되었습니다. 이 첫 번째 연구 물결은 IBM System S, Esper, Oracle CQL/CEP 및 TIBCO와 같이 다음 해(대략 2004~2010년)에 개발된 상용 스트림 처리 시스템에 큰 영향을 미쳤습니다. 이러한 시스템은 대부분 스트리밍 창 쿼리와 CEP(복합 이벤트 처리)에 중점을 두었습니다. 이 시스템 시대는 주로 확장 아키텍처, 정렬된 이벤트 스트림 처리가 특징입니다.

B 아스테리오스 카트시포디모스  
a.katsifodimos@tudelft.nl

마리オス 프라그콜리  
s.marios.fragkoulis@deliveryhero.com

파리카본  
parisc@kth.se ; paris.carbone@ri.se

바실리키 칼라브리  
vkalavri@bu.edu

<sup>1</sup> Delivery Hero Research, 베를린, 독일

<sup>2</sup> 델프트 공과대학교, 델프트, 네덜란드

<sup>3</sup> KTH 왕립공과대학, 스웨덴 스톡홀름

<sup>4</sup> RISE, 스톡홀름, 스웨덴

<sup>5</sup> 보스턴 대학교, 보스턴, 미국

2세대 스트리밍 시스템은 대략 MapReduce의 도입 [61]과 클라우드 컴퓨팅의 대중화 이후 시작된 연구의 결과였다. 분산형 데이터 병렬 처리 엔진과 상용 하드웨어의 비공유 아키텍처뿐만 아니라 주류 MapReduce와 유사한 시스템을 지원할 수 있는 시스템 설계로 초점이 옮겨졌습니다.



# A survey on the evolution of stream processing systems

Marios Fragkoulis<sup>1</sup> · Paris Carbone<sup>3,4</sup> · Vasiliki Kalavri<sup>5</sup> · Asterios Katsifodimos<sup>2</sup>

Received: 10 August 2022 / Revised: 1 September 2023 / Accepted: 11 September 2023 / Published online: 22 November 2023  
© The Author(s) 2023

## Abstract

Stream processing has been an active research field for more than 20 years, but it is now witnessing its prime time due to recent successful efforts by the research community and numerous worldwide open-source communities. This survey provides a comprehensive overview of fundamental aspects of stream processing systems and their evolution in the functional areas of out-of-order data management, state management, fault tolerance, high availability, load management, elasticity, and reconfiguration. We review noteworthy past research findings, outline the similarities and differences between the first ('00–'10) and second ('11–'23) generation of stream processing systems, and discuss future trends and open problems.

**Keywords** Stream processing · Fault-tolerance · Streaming analytics · Cloud applications

## 1 Introduction

Applications of stream processing technology have gone through a resurgence, penetrating multiple and very diverse industries. Nowadays, virtually all Cloud vendors offer first-class support for deploying managed stream processing pipelines, while streaming systems are used in a variety of use-cases that go beyond the classic streaming analytics (windows, aggregates, joins, etc.). Some examples include dynamic car-trip pricing, credit card fraud detection, predictive analytics, monitoring, and real-time traffic control. At the moment of writing, we are witnessing a trend towards using stream processors to build more general event-driven architectures [96], large-scale continuous ETL and analytics, and microservices [91].

✉ Asterios Katsifodimos  
a.katsifodimos@tudelft.nl

Marios Fragkoulis  
marios.fragkoulis@deliveryhero.com

Paris Carbone  
parisc@kth.se ; paris.carbone@ri.se

Vasiliki Kalavri  
vkalavri@bu.edu

<sup>1</sup> Delivery Hero Research, Berlin, Germany

<sup>2</sup> Delft University of Technology, Delft, The Netherlands

<sup>3</sup> KTH Royal Institute of Technology, Stockholm, Sweden

<sup>4</sup> RISE, Stockholm, Sweden

<sup>5</sup> Boston University, Boston, USA

During the last 20 years, streaming technology has evolved significantly, under the influence of database and distributed systems. The notion of streaming queries was first introduced in 1992 by the Tapestry system [148], and was followed by lots of research on stream processing in the early 00s. Fundamental concepts and ideas originated in the database community and were implemented in prototype systems such as TelegraphCQ [48], Stanford's STREAM, NiagaraCQ [51], Auroral/Borealis [12], and Gigascope [54]. Although these prototypes roughly agreed on the data model, they differed considerably on querying semantics [21, 33]. This research period also introduced various systems challenges, such as sliding-window aggregation [22, 107], fault-tolerance and high availability [30, 137], as well as load balancing and shedding [144]. This first wave of research was highly influential on commercial stream processing systems that were developed in the following years (roughly during 2004 – 2010), such as IBM System S, Esper, Oracle CQL/CEP and TIBCO. These systems focused—for the most part—on streaming window queries and complex event processing (CEP). This era of systems was mainly characterized by scale-up architectures, processing ordered event streams.

The second generation of streaming systems was a result of research that started roughly after the introduction of MapReduce [61] and the popularization of Cloud Computing. The focus shifted towards not only distributed, data-parallel processing engines and shared-nothing architectures on commodity hardware, but also on the design of systems that can support the mainstream MapReduce-like

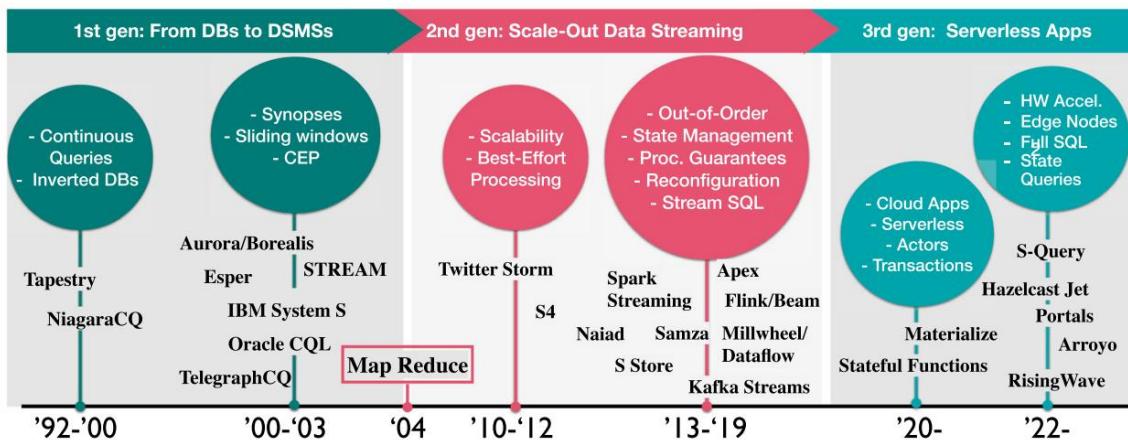


그림 1. 스트림 처리 시스템의 진화와 각 초기 영역에 대한 개요

사용자 정의 함수(UDF) 기반 프로그래밍 모델.

결과적으로 Millwheel [14], Storm [3], Spark 와 같은 시스템 Streaming [164] 및 Apache Flink [37]는 스트리밍 계산을 하드 코딩된 방식으로 표현하기 위해 처음으로 노출된 프리미티브입니다.

데이터 흐름 그래프와 분산 클러스터에서 투명하게 처리되는 데이터 병렬 실행.

Google Dataflow 모델

[16] 비순차적 처리 [108] 및 구두점 [155] 과 같은 오래된 아이디어를 다시 도입하여 통일된

스트리밍 및 배치 계산을 위한 병렬 처리 모델입니다. 이 시대의 스트림 프로세서는 다음으로 수렴되고 있습니다.

대규모 비순차적 장애에 대한 내결합성, 확장형 처리 스트림.

그림 1은 영향력 있는 스트리밍 시스템을 3세대 및 주요 특징으로 도식적으로 분류한 것입니다.

각 시대의 초기 영역. 비록 기초는

스트림 처리는 크게 변하지 않았습니다.

수년 동안 스트림 처리 시스템은 다음과 같이 변화했습니다.

정교하고 확장 가능한 엔진으로 올바른 결과 생성

실패가 있는 경우. 초기 시스템 및 언어

관계형 실행 엔진의 확장으로 설계되었습니다.

창문이 추가되었습니다. 최신 스트리밍 시스템

완전성과 가치에 대해 추론하는 방식으로 진화해 왔습니다.

순서를 정하고(예: 순서가 잘못된 계산) 목격한 적이 있습니다.

처리 보장, 재구성 및 상태의 기반을 구성하는 아키텍처 패러다임 전환

관리. 글을 쓰는 순간 우리는 새로운 것을 관찰합니다

데이터 통합에 초점을 맞춘 스트리밍 시스템의 변형

클라우드 서비스 및 서비스 앱을 사용한 스트리밍 또는

엣지 컴퓨팅 및 특수 하드웨어. IoT의 요구 사항

엣지 컴퓨팅으로 인해 더 많은 제품이 창출되었습니다.

하드웨어 인식 [167] 및 리소스 사용량이 적은 시스템 [72]

이벤트 기반 파이프라인의 경우. 동시에 추가 기능으로서 전체 SQL 지원을 향한 경향이 관찰되었습니다.

기존 시스템에서 [159] 뿐만 아니라 새로운 클라우드의 기반으로도 사용

스트리밍 데이터베이스 시스템 [4, 6, 8]. 이벤트 기반 클라우드와 서비스 시스템

분야에서는 현재

액터와 데이터 스트리밍 패러다임의 통합,

행위자의 유연성과 보증을 결합하고

스트리밍 엔진의 성능 [2, 13, 34, 140].

스트리밍 처리 시스템의 진화는 [39, 59, 78] 이전에 연구 커뮤니티에 관심을 가져왔습니다. 이것

설문조사는 체계적이고

현장의 현황을 제시하기보다는 기본 기능 영역의 진화에 대한 포괄적인 조사

특정 시점에. 우리가 아는 한에,

이는 또한 근본적인 이해를 위한 첫 번째 시도이기도 하다.

특정 초기 기술과 디자인이 널리 퍼진 이유

현대 시스템에서는 다른 시스템은 버려졌습니다. 또한,

아이디어가 어떻게 살아남고, 진화하고, 자주 재발명되었는지 조사하면서 다양한 용어를 사용하는 용어를 조정합니다.

스트리밍 시스템 세대.

## 1.1 기여

본 설문지를 통해 우리는 다음과 같은 기여를 합니다.

- 스트리밍 시스템에 대한 기준 접근 방식을 요약합니다.

기본 가정 및 메커니즘 측면에서 초기 및 최신 스트리밍 프로세서를 설계하고 분류합니다.

- 비순차적 데이터 관리, 상태와 관련하여 초기 및 현대 스트리밍 처리 시스템을 비교합니다.

관리, 내결합성, 고가용성, 로드 관리, 탄력성 및 재구성.

- 우리는 영향을 준 초기 작품을 강조합니다.

오늘날의 스트리밍 시스템 디자인.

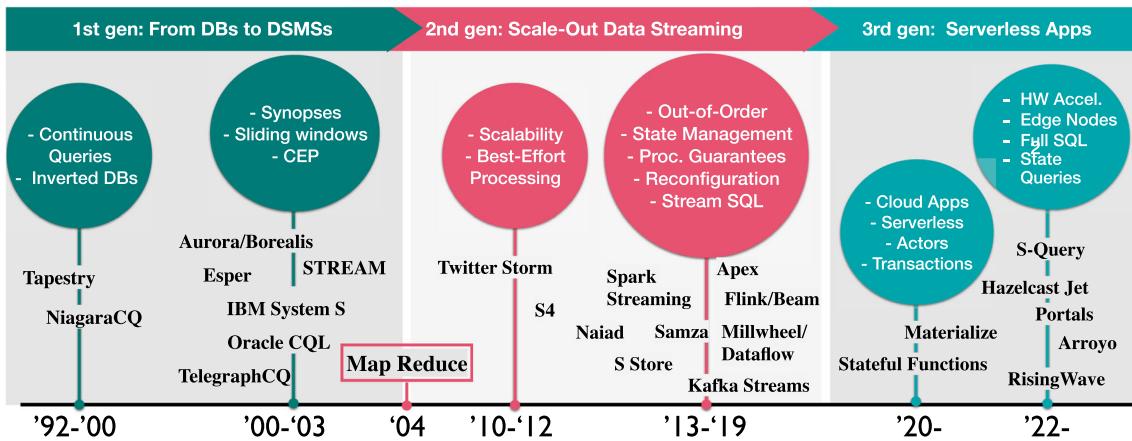
- 우리는 기본에 대한 공통 명명법을 설정합니다.

일관성이 없는 것으로 종종 설명되는 스트리밍 개념

다양한 시스템과 커뮤니티의 용어.

- 스트리밍 가능성에 대한 정교한 정의를 제공합니다.

처리 시스템.



**Fig. 1** An overview of the evolution of stream processing systems and respective domains of focus

user-defined function (UDF)-based programming models. As a result, systems such as Millwheel [14], Storm [3], Spark Streaming [164], and Apache Flink [37] first exposed primitives for expressing streaming computations as hard-coded dataflow graphs and transparently handled data-parallel execution on distributed clusters. The Google Dataflow model [16] re-introduced older ideas such as out-of-order processing [108] and punctuations [155], proposing a unified parallel processing model for streaming and batch computations. Stream processors of this era are converging towards fault-tolerant, scale-out processing of massive out-of-order streams.

Figure 1 presents a schematic categorization of influential streaming systems into three generations and highlights each era's domains of focus. Although the foundations of stream processing have remained largely unchanged over the years, stream processing systems have transformed into sophisticated and scalable engines, producing correct results in the presence of failures. Early systems and languages were designed as extensions of relational execution engines, with the addition of windows. Modern streaming systems have evolved in the way they reason about completeness and ordering (e.g., out-of-order computation) and have witnessed architectural paradigm shifts that constituted the foundations of processing guarantees, reconfiguration, and state management. At the moment of writing, we observe new variants of streaming systems focusing on integrating data streaming with either cloud services and serverless apps or edge computing and specialized hardware. The needs of IoT and edge computing have attracted the creation of more hardware-aware [167] and less resource-heavy systems [72] for event-based pipelines. At the same time, we observe a tendency towards full SQL support, both as an add-on capability in older systems [159], but also as a foundation for new cloud streaming database systems [4, 6, 8]. In the space of event-based cloud and serverless systems, we currently witness

an integration of the actor and data-streaming paradigms, combining the flexibility of actors with the guarantees and performance of streaming engines [2, 13, 34, 140].

The evolution of stream processing systems has concerned the research community before [39, 59, 78]. This survey extends prior work by providing a systematic and comprehensive investigation of the evolution of fundamental functional areas rather than presenting the state of the field at a particular point in time. To the best of our knowledge, this is also the first attempt at understanding the underlying reasons why certain early techniques and designs prevailed in modern systems while others were abandoned. Further, by examining how ideas survived, evolved, and were often re-invented, we reconcile the terminology used by the different generations of streaming systems.

## 1.1 Contributions

With this survey paper, we make the following contributions:

- We summarize existing approaches to streaming systems design and categorize early and modern stream processors in terms of underlying assumptions and mechanisms.
- We compare early and modern stream processing systems with regard to out-of-order data management, state management, fault-tolerance, high availability, load management, elasticity, and reconfiguration.
- We highlight foundational works that have influenced today's streaming systems design.
- We establish a common nomenclature for fundamental streaming concepts, often described by inconsistent terms in different systems and communities.
- We provide a refined definition of availability for stream processing systems.

## 1.2 관련 조사 및 연구 모음

우리는 다음 설문조사를 우리 설문조사를 보완하는 것으로 보고 스트림 처리의 특정 측면에 대해 더 깊이 파고드는 데 관심이 있거나 스트리밍 기술과 인접 연구 커뮤니티의 발전을 비교하려는 독자에게 권장합니다.

Cugola 및 Margara [55]는 활성 데이터베이스 및 복합 이벤트 처리 시스템과 같은 관련 기술과 관련된 스트림 처리에 대한 관점을 제공하고 데이터 스트리밍 시스템과의 관계를 논의합니다. 또한 스트리밍 언어와 스트리밍 연산자 의미 체계의 분류를 제공합니다. 언어 측면은 매우 큰 데이터 스트림의 문제를 해결하기 위해 개발된 언어에 초점을 맞춘 또 다른 최근 조사 [82]에서 더 자세히 다루고 있습니다. 데이터 모델, 실행 모델, 도메인 및 의도된 사용자 대상 측면에서 스트리밍 언어의 특성을 지정합니다. Röger와 Mayer [134]는 스트리밍 시스템의 병렬화 및 탄력성 접근 방식에 대한 최근 연구의 개요를 제시합니다. 그들은 연산자 병렬화 전략과 병렬성 적용 방법을 도입하는 데 사용하는 일반 시스템 모델을 정의합니다. 그들의 분석은 또한 다양한 연구 커뮤니티에서 시작된 탄력성 접근 방식을 비교하는 것을 목표로 합니다. Hirzelet et al. [83]은 스트리밍 쿼리 계획에 대한 논리적 및 물리적 최적화의 광범위한 목록을 제시합니다. 가정, 의미, 적용 가능성 시나리오 및 절충 측면에서 스트리밍 최적화의 분류를 제시합니다. 또한 수익성을 추론하기 위한 실험적 증거를 제시하고 시스템 구현자가 적절한 최적화를 선택하도록 안내합니다. To, Soto 및 Markl [150]은 빅 데이터 관리 시스템에서 상태 개념과 적용을 조사하고 스트리밍 상태 측면을 다룹니다. 마지막으로 Dayarathna와 Perera [58]는 시스템 아키텍처, 사용 사례 및 뜨거운 연구 주제에 중점을 두고 지난 10년간의 발전에 대한 조사를 제시합니다. 지원하는 작업 유형, 내결합성 기능, 프로그래밍 언어 사용 및 보고된 최고 성능과 같은 기능 측면에서 최신 시스템을 요약합니다.

우리는 독자에게 Garofalakis et al. [69] 스트리밍 데이터 관리의 이론적 기초, 스트리밍 알고리즘 등 관련 주제를 포괄적으로 다루고 있습니다. 이 컬렉션은 1세대 스트리밍 시스템의 주요 기여에 중점을 두고 있습니다. 기본 알고리즘 및 개요, 스트리밍 데이터 마이닝의 기본 결과, 스트리밍 언어 및 연산자 의미 체계, 다양한 도메인의 대표적인 애플리케이션 세트를 검토합니다.

## 1.3 조사기관

우리는 섹션에서 도메인의 필수 요소를 제시하는 것으로 시작합니다. 2. 그런 다음 스트림 처리 시스템이 제공하는 중요한 기능 각각에 대해 자세히 설명합니다.

주문 데이터 관리(섹션 3), 상태 관리(섹션 4), 내결합성 및 고가용성(섹션 5), 로드 관리, 탄력성 및 재구성(섹션 6). 각 섹션에는 초기 접근 방식과 현대 접근 방식을 비교하는 1세대 및 2세대 토론과 공개 문제 요약이 포함되어 있습니다. 곧 7에서는 이러한 각 차원에 따라 하천 시스템에 대한 설계 고려 사항을 요약하고 주요 결과를 강조하며 향후 전망에 대해 논의합니다.

요약 표 대략적으로 모든 섹션에서 이 설문 조사는 일련의 시스템과 각 섹션에서 논의된 시스템의 특성 또는 특징을 포함하는 요약 표를 제공합니다. 각 섹션에서 고려하고 서로 비교하는 시스템 세트는 다를 수 있습니다. 이러한 차이점은 시스템(또는 논문)이 특정 특성을 지원하지 않을 수 있기 때문입니다. 예를 들어 Apache Storm은 타임스탬프나 이벤트 도착 순서를 고려하지 않으므로 표 1에 포함되지 않습니다.

## 2 예선

이 섹션에서는 필요한 배경 정보를 제공하고 이 설문 조사의 나머지 부분이 의존하는 기본적인 스트림 처리 개념을 설명합니다. 스트리밍 시스템의 주요 요구 사항을 논의하고 기본 스트리밍 데이터 모델을 소개하며 초기 및 최신 스트리밍 시스템의 아키텍처에 대한 높은 수준의 개요를 제공합니다.

### 2.1 스트리밍 시스템 요구사항

데이터 스트림은 처리가 시작되기 전에 전체를 사용할 수 있는 것이 아니라 시간이 지남에 따라 점진적으로 생성되는 데이터 세트입니다 [69]. 데이터 스트림은 제한이 없는 실시간 데이터입니다. 따라서 스트림 처리 시스템은 액세스 가능한 방식으로 전체 스트림을 저장할 수 없으며 데이터 도착 속도나 순서를 제어할 수도 없습니다. 기존 데이터 관리 인프라와 달리 스트리밍 시스템은 제한된 메모리를 사용하여 요소를 즉시 처리해야 합니다. 스트림 요소는 지속적으로 도착하며 소스 또는 도착 시 할당될 수 있는 타임 스탬프를 하나 이상 포함합니다.

스트리밍 쿼리는 이벤트를 수집하고 데이터에 대한 단일 패스 또는 제한된 수의 패스를 사용하여 지속적인 방식으로 결과를 생성합니다. 스트리밍 쿼리 처리는 여러 가지 이유로 어렵습니다. 첫째, 업데이트된 결과를 지속적으로 생성하려면 지금까지 본 스트림에 대한 기록 정보를 효율적으로 쿼리하고 업데이트할 수 있는 압축된 표현으로 저장해야 할 수 있습니다. 이러한 요약 표현을 스케치 또는 개요라고 합니다. 둘째, 높은 입력 속도를 처리하기 위해 특정 쿼리에서는 인덱스 및 구체화된 뷰를 지속적으로 업데이트할 여유가 없을 수 있습니다. 셋째, 스트림 프로세서는 가정에 의존할 수 없습니다.

## 1.2 Related surveys and research collections

We view the following surveys as complementary to ours and recommend them to readers interested in diving deeper into a particular aspect of stream processing or those who seek a comparison between streaming technology and advances from adjacent research communities.

Cugola and Margara [55] provide a view of stream processing with regard to related technologies, such as active databases and complex event processing systems, and discuss their relationship with data-streaming systems. Further, they provide a categorization of streaming languages and streaming-operator semantics. The language aspect is further covered in another recent survey [82], which focuses on the languages developed to address the challenges in very large data streams. It characterizes streaming languages in terms of data model, execution model, domain, and intended user audience. Röger and Mayer [134] present an overview of recent work on parallelization and elasticity approaches of streaming systems. They define a general system model, which they use to introduce operator-parallelization strategies and parallelism-adaptation methods. Their analysis also aims at comparing elasticity approaches originating in different research communities. Hirzel et al. [83] present an extensive list of logical and physical optimizations for streaming query plans. They present a categorization of streaming optimizations in terms of their assumptions, semantics, applicability scenarios, and trade-offs. They also present experimental evidence to reason about profitability and guide system implementers in selecting appropriate optimizations. To, Soto, and Markl [150] survey the concept of state and its applications in big data management systems, also covering aspects of streaming state. Finally, Dayarathna and Perera [58] present a survey of the advances of the last decade with a focus on system architectures, use-cases, and hot research topics. They summarize recent systems in terms of their features, such as what types of operations they support, their fault-tolerance capabilities, their use of programming languages, and their best reported performance.

We refer the reader to Garofalakis et al. [69] for a comprehensive coverage of related topics, such as theoretical foundations of streaming data management and streaming algorithms. That collection focuses on major contributions of the first generation of streaming systems. It reviews basic algorithms and synopses, fundamental results in stream data mining, streaming languages and operator semantics, and a set of representative applications from different domains.

## 1.3 Survey organization

We begin by presenting the essential elements of the domain in Sect. 2. Then we elaborate on each of the important functionalities offered by stream processing systems: out-of-

order data management (Sect. 3), state management (Sect. 4), fault tolerance and high availability (Sect. 5), and load management, elasticity, and reconfiguration (Sect. 6). Each one of these sections contains a *first-generation vs. second-generation* discussion that compares early to contemporary approaches, and a summary of open problems. In Sect. 7, we summarize design considerations for stream systems along each of these dimensions, we highlight our major findings, and we discuss future prospects.

**Summarizing tables** Roughly in every section, this survey provides a summarizing table, containing a set of systems, as well as their characteristics or features discussed by the respective section. Note that the set of systems that are considered and compared against each other in each section may differ. This difference is because a system (or paper) may not support a certain characteristic. For instance, Apache Storm is not part of Table 1, as it does not consider timestamps or order of event arrivals.

## 2 Preliminaries

In this section, we provide necessary background and explain fundamental stream processing concepts on which the rest of this survey relies. We discuss the key requirements of a streaming system, introduce the basic streaming data models, and give a high-level overview of the architecture of early and modern streaming systems.

### 2.1 Requirements of streaming systems

A data stream is a data set that is produced incrementally over time, rather than being available in full before its processing begins [69]. Data streams are real-time data that might be unbounded. Therefore, stream processing systems can neither store the entire stream in an accessible way nor can they control the data arrival rate or order. In contrast to traditional data-management infrastructure, streaming systems have to process elements on-the-fly using limited memory. Stream elements arrive continuously and bear at least one timestamp, which can be assigned at the source or on arrival.

A streaming query ingests events and produces results in a continuous manner, using a single pass or a limited number of passes over the data. Streaming query processing is challenging for multiple reasons. First, continuously producing updated results might require storing historical information about the stream seen so far in a compact representation that can be queried and updated efficiently. Such summary representations are known as *sketches* or *synopses*. Second, in order to handle high input rates, certain queries might not afford to continuously update indexes and materialized views. Third, stream processors cannot rely on the assumption

연관된 입력으로부터 상태를 재구성할 수 있다는 점입니다.

수명이 짧고 고정 크기 입력을 처리하는 일괄 쿼리와 달리 스트리밍 쿼리는 오랫동안 실행됩니다. 결과적으로 중간 상태를 재구성하려면 전체 스트림 기록을 다시 처리해야 할 수도 있습니다. 대신, 허용 가능한 성능을 달성하기 위해 스트리밍 운영자는 충분 계산을 활용해야 합니다.

#### 앞서 언급한 데이터 스트림의 특성과

연속 쿼리는 짧은 대기 시간과 높은 처리량이라는 확실한 성능 요구 사항 외에 스트리밍 시스템에 대한 일련의 고유한 요구 사항을 제공합니다. 입력 순서에 대한 제어가 부족하기 때문에 스트리밍 시스템은 순서가 잘못되고 지연된 데이터를 수신할 때 올바른 결과를 생성해야 합니다(참조):

분파. 삼). 스트림의 진행 상황과 결과 완전성에 대한 이유를 추정하는 메커니즘을 구현해야 합니다. 또한 스트리밍 쿼리의 장기 실행 특성으로 인해 스트리밍 시스템은 누적된 상태를 관리하고(섹션 4 참조) 오류로부터 이를 보호해야 합니다(섹션 5 참조). 마지막으로, 데이터 입력 속도를 제어할 수 없으면 성능 저하 없이 작업 부하 변화를 처리할 수 있도록 적응형 스트림 프로세서가 필요합니다(섹션 6 참조).

#### 2.2.2 데이터플로우 스트리밍 모델

데이터 흐름 스트리밍 모델은 데이터 흐름 그래프, 즉 방향성 그래프  $G = (E, V)$ 로 표현됩니다. 여기서  $V$ 의 정점은 연산자를 나타내고  $E$ 의 가장자리는 데이터 스트림을 나타냅니다. 2세대 시스템 [16, 37, 164]에 의해 구현된 데이터 흐름 스트리밍 모델은 타임스탬프의 존재 외에는 입력 스트림 요소에 엄격한 스키마나 의미를 부과하지 않습니다. Naiad [120]와 같은 일부 시스템에서는 모든 스트림 요소에 논리적 타임스탬프가 있어야 하지만 Flink [37] 및 Dataflow [16]와 같은 다른 시스템에서는 시간 영역 선언이 필요합니다. 애플리케이션은 세 가지 모드 중 하나로 작동할 수 있습니다. (i) 이벤트(또는 애플리케이션) 시간은 소스에서 이벤트가 생성되는 시간, (ii) 수집 시간은 이벤트가 시스템에 도착하는 시간, (iii) 처리 시간입니다. 스트리밍 시스템에서 이벤트가 처리되는 시간입니다. 최신 데이터 흐름 스트리밍 시스템은 해당 요소가 추가, 삭제, 교체 또는 델타를 나타내는지 여부에 관계없이 모든 유형의 입력 스트림을 수집할 수 있습니다. 애플리케이션 개발자는 의미 체계를 적용하고 이에 따라 상태를 업데이트하고 올바른 결과를 생성하는 연산자 논리를 작성하는 일을 담당합니다. 일반적으로 수집 시에는 키와 값을 지정할 필요가 없습니다. 그러나 특정 데이터 병렬 연산자를 사용할 때는 키를 정의해야 합니다.

## 2.2 스트리밍 데이터 모델

많은 이론적 스트리밍 데이터 모델이 존재하며 주로 스트리밍 알고리즘의 공간 요구 사항과 계산 복잡성을 연구하고 어떤 스트리밍 계산이 실용적인지 이해하는 목적으로 사용됩니다.

예를 들어 스트림은 동적 1차원 벡터로 모델링될 수 있습니다 [69]. 모델은 스트림의 새 요소를 사용할 수 있을 때 이 동적 벡터가 업데이트되는 방법을 정의합니다. 이론적 스트리밍 데이터 모델은 알고리즘 설계에 유용하지만 초기 스트림 처리 시스템은 대신 관계형 데이터 모델의 확장을 채택했습니다.

최근 스트리밍 데이터 흐름 시스템, 특히 MapReduce 철학의 영향을 받은 시스템은 데이터 스트림 모델링의 책임을 애플리케이션 개발자에게 맡깁니다.

#### 2.2.1 관계형 스트리밍 모델

1세대 시스템 [11, 12, 20, 31, 48, 54]에 의해 구현된 관계형 스트리밍 모델에서 스트림은 공통 스키마에 대한 변화하는 관계를 설명하는 것으로 해석됩니다. 특히 이러한 시스템은 스트림 요소에 타임스탬프 또는 시퀀스 번호가 포함되어 스트림에 대한 창을 정의할 수 있다고 가정했습니다. 스트림 자체는 외부 소스 및 업데이트 관계 테이블에 의해 생성되거나 지속적인 쿼리 및 업데이트 구체화된 뷰에 의해 생성됩니다. 연산자는 연산자의 관계형 의미에 따라 입력 스트림에 대해 계산된 변경 뷰를 설명하는 이벤트 스트림을 출력합니다. 따라서 의미론과 관계 스키마 모두 시스템에 의해 부과됩니다.

## 3 행사순서 및 적시성 관리

이벤트 순서는 일반적으로 이벤트 타임스탬프를 사용하여 적용됩니다. 이벤트 타임스탬프는 각 레코드에 포함되어 있으며 레코드의 데이터가 소스에서 생성된 이벤트 시간을 나타냅니다. 이벤트 타임스탬프는 스트림의 데이터 순서를 지정하며 스트림 의미의 일부로 간주됩니다 [113]. 수행할 계산에 따라 스트리밍 시스템은 의미상 올바른 결과를 제공하기 위해 스트림 레코드를 특정 순서로 처리해야 할 수도 있습니다 [142]. 그러나 일반적인 경우 스트림의 레코드는 Sect. 3.1.

순서가 잘못된 데이터 레코드 [142, 153]는 이후 이벤트 타임스탬프가 있는 레코드 이후 입력 소스에서 스트리밍 시스템에 도착합니다.

논문의 나머지 부분에서는 하천 데이터 기록의 질서 교란을 지칭하기 위해 무질서 [113]와 비순차 [14, 108]라는 용어를 같은 의미로 사용합니다. 질서에 대한 추론과 무질서를 관리하는 것은 스트리밍 시스템 운영에 있어서 기본적인 고려사항이다.

우리는 섹션에서 장애의 원인을 강조합니다. 3.1 절에서는 하천 기록의 무질서와 처리 진행 사이의 관계를 명확히 합니다. 3.2, 그리고 Sect.에서 비순차적 데이터를 관리하기 위한 두 가지 주요 시스템 아키텍처를 간략하게 설명합니다. 3.3. 그런 다음 섹션에서 무질서의 결과를 설명합니다. 3.4 및 장애 관리를 위한 현재 메커니즘

tion that state can be reconstructed from associated inputs. In contrast to batch queries that are short-lived and process fixed-size inputs, streaming queries are long-running. As a result, reconstructing their intermediate state may require re-processing the entire stream history. Instead, to achieve acceptable performance, streaming operators need to leverage incremental computation.

The aforementioned characteristics of data streams and continuous queries provide a set of unique requirements for streaming systems, other than the evident performance ones of low latency and high throughput. Given the lack of control over the input order, a streaming system needs to produce correct results when receiving out-of-order and delayed data (cf. Sect. 3). It needs to implement mechanisms that estimate a stream's progress and reason about result completeness. Further, the long-running nature of streaming queries demands that streaming systems manage accumulated state (cf. Sect. 4) and guard it against failures (cf. Sect. 5). Finally, having no control over the data input rate requires stream processors to be adaptive so that they can handle workload variations without sacrificing performance (cf. Sect. 6).

## 2.2 Streaming data models

There exist many theoretical streaming data models, mainly serving the purpose of studying the space requirements and computational complexity of streaming algorithms and understanding which streaming computations are practical. For instance, a stream can be modeled as a dynamic one-dimensional vector [69]. The model defines how this dynamic vector is updated when a new element of the stream becomes available. While theoretical streaming data models are useful for algorithm design, early stream processing systems instead adopted extensions of the *relational* data model. Recent streaming dataflow systems, especially those influenced by the MapReduce philosophy, place the responsibility of data stream modeling on the application developer.

### 2.2.1 Relational streaming model

In the relational streaming model, as implemented by first-generation systems [11, 12, 20, 31, 48, 54], a stream is interpreted as describing a changing relation over a common schema. Notably, these systems assumed that stream elements bear timestamps or sequence numbers, allowing for defining windows over streams. Streams themselves are either produced by external sources and update relation tables or are produced by continuous queries and update materialized views. An operator outputs event streams that describe the changing view computed over the input stream according to the relational semantics of the operator. Thus, both the semantics and the relation schema are imposed by the system.

### 2.2.2 Dataflow streaming model

The dataflow streaming model is represented as a dataflow graph, that is, a directed graph  $G = (E, V)$ , where vertices in  $V$  represent operators and edges in  $E$  denote data streams. The dataflow streaming model, as implemented by systems of the second generation [16, 37, 164], does not impose any strict schema or semantics on the input stream elements, other than the presence of a timestamp. While some systems, such as Naiad [120], require that all stream elements bear a logical timestamp, other systems, such as Flink [37] and Dataflow [16], expect the declaration of a *time domain*. Applications can operate in one of three modes: (i) *event* (or application) time is the time when events are generated at the sources, (ii) *ingestion* time is the time when events arrive at the system, and (iii) *processing* time is the time when events are processed in the streaming system. Modern dataflow streaming systems can ingest any type of input stream, irrespectively of whether its elements represent additions, deletions, replacements or deltas. The application developer is responsible for imposing the semantics and writing the operator logic to update state accordingly and produce correct results. Designating keys and values is also usually not required at ingestion time, however, keys must be defined when using certain data-parallel operators.

## 3 Managing event order and timeliness

Event order is typically enforced using an event timestamp. An event timestamp is contained in each record and denotes the event time when the record's data were generated at the source. The event timestamps dictate the order of data in the stream and they are considered part of the stream's semantics [113]. Depending on the computations to perform, a streaming system may have to process stream records in a certain order to provide semantically correct results [142]. However, in the general case, a stream's records arrive out of order [104, 155] for reasons explained in Sect. 3.1.

*Out-of-order* data records [142, 153] arrive at a streaming system from an input source after records with later event time timestamps.

In the rest of the paper, we use the terms *disorder* [113] and *out-of-order* [14, 108] interchangeably to refer to the disturbance of order in a stream's data records. Reasoning about order and managing disorder are fundamental considerations for the operation of streaming systems.

We highlight the causes of disorder in Sect. 3.1, we clarify the relationship between disorder in a stream's records and processing progress in Sect. 3.2, and we outline the two key system architectures for managing out-of-order data in Sect. 3.3. Then, we describe the consequences of disorder in Sect. 3.4 and present mechanisms for managing disorder in

분파. 3.5. 마지막으로 섹션에서. 3.6에서는 초기 시스템과 현대 시스템의 비순차적 데이터 관리의 차이점을 논의하고 Sect.에서 열린 문제를 제시합니다. 3.7.

### 3.1 장애의 원인

데이터 스트림의 장애는 스트리밍 시스템 외부의 확률적 요인이나 시스템 내부에서 발생하는 작업으로 인해 발생할 수 있습니다.

#### 장애를 일으키는 가장 흔한 외부 요인

스트림은 네트워크 [99, 142]입니다. 네트워크의 신뢰성, 대역폭 및 로드에 따라 일부 스트림 레코드의 라우팅은 다른 레코드의 라우팅에 비해 완료하는데 시간이 더 오래 걸릴 수 있으며 이로 인해 스트리밍 시스템의 이벤트 시간 측면에서 도착 순서가 달라질 수 있습니다. 개별 스트림의 레코드 순서가 유지되더라도 동기화된 시계를 사용하더라도 센서와 같은 여러 소스에서 수집하면 일반적으로 무질서한 레코드 수집이 발생합니다.

특히 소스에 동기화된 시계 기능이 없는 경우(종종 발생함) 생성된 레코드의 타임스탬프가 실제 제작 시간과 일치하지 않을 수 있습니다. 결과적으로, 순서를 지정하는 데 사용된 타임스탬프가 이벤트 시간을 나타내지 않으면 이러한 장애 원인을 수정할 수 없습니다.

외부 요인은 제쳐두고 스트림의 특정 작업으로 인해 기록 순서가 깨집니다. 먼저, 조인 처리는 두 개의 스트림을 취하고 둘의 섞인 조합을 생성합니다. 왜냐하면 병렬 조인 연산자가 조인 속성에 따라 데이터를 다시 분할하고 [157] 일치 순서에 따라 조인 결과를 출력하기 때문입니다 [77, 90]. 둘째, 순서 속성과 다른 속성을 기반으로 한 윈도우ing은 스트림을 재정렬합니다 [54].셋째, 순서와 다른 속성을 사용하여 데이터 우선순위 [130, 156]를 지정하면 스트림의 순서도 변경됩니다. 마지막으로, 동기화되지 않은 두 스트림에 대한 결합 연산을 통해 두 입력 스트림의 모든 레코드가 무작위 순서로 서로 인터리브되는 스트림이 생성됩니다 [12].

### 3.2 장애 및 처리과정

장애를 관리하기 위해 스트리밍 시스템은 처리 진행 상황을 감지하고 측정해야 합니다. 진행 상황은 시간이 지남에 따라 스트림 레코드 처리가 얼마나 진행되었는지를 나타냅니다. 처리 진행 상황은 스트림을 주문하는 스트림 레코드의 속성 A를 사용하여 정의되고 수량화될 수 있습니다. 시간이 지남에 따라 처리되지 않은 레코드 중 가장 작은 A 값이 증가하면 스트림 처리가 진행됩니다 [108]. A then은 진행 중인 속성이고 A 자체의 가장 작은(처리되지 않은) 값은 시스템이 처음부터 기록 처리에서 얼마나 멀리 도달했는지 나타내기 때문에 진행의 척도입니다. 처리 진행 상황은 스트림 레코드의 둘 이상의 속성을 사용하여 수량화할 수 있습니다.

처리 진행 상황을 추적하는 스트리밍 시스템의 기능을 통해 시스템은 처리 진행 중에 스트리밍 시스템이 허용할 수 있는 장애 수준을 정량화하기 위해 자연 한계를 결정할 수 있습니다. 자세히 설명하자면, 자연 경계는 나머지 스트림과 관련하여 스트리밍 시스템에서 처리하기 위해 데이터가 얼마나 늦게 받아들여질 수 있는지를 측정하는 것입니다. 일반적으로 시간 측정 또는 스트림 레코드 수에 해당합니다. 예를 들어 자연 범위가 1초이면 스트리밍 시스템의 창 연산자가 창의 시간 범위보다 1초 늦게 창의 기능을 계산하게 됩니다. 시간 확장을 사용하면 순서가 잘못된 레코드를 창 계산에 포함할 수 있습니다.

### 3.3 장애 관리를 위한 시스템 아키텍처

두 가지 주요 아키텍처 원형은 장애 관리와 관련하여 스트리밍 시스템 설계에 영향을 미쳤습니다. (i) 순차 처리 시스템 [12, 21, 54, 142] 및 (ii) 비순차 처리 시스템 [14, 37, 108, 120].

순차 처리 시스템은 세 가지 주요 전략 중 하나를 사용하여 장애를 관리합니다. 첫째, 그들은 입력 스트림이 정렬되어 있다고 가정하고 늦은 데이터를 버릴 수 있습니다 [12, 21, 54]. 둘째, 스트림 순서를 적용하기 위해 자연 한계까지 입력 스트림을 버퍼링하고 재정렬할 수 있습니다 [12, 21, 142]. 그런 다음 처리를 위해 재정렬된 레코드를 전달하고 해당 버퍼를 지웁니다. 마지막으로 자연 한계까지 재정렬하지 않고 자연된 데이터를 승인할 수 있습니다 [12]. 마지막 전략은 무질서한 입력을 지원하지만 다운스트림 운영자도 독립적으로 자연 한계를 정량화하고 시행해야 합니다. 이를 효과적으로 수행하려면 처리 진행 상황을 추적해야 하는데, 이는 시스템 전체의 진행 상황 추적 메커니즘 없이는 달성하기 어렵습니다. 따라서 순차 시스템은 일반적으로 스트림 순서를 적용 및 보존하고 이를 사용하여 처리 진행 상황을 추론합니다.

비순차적 처리 시스템에서 운영자 또는 글로벌 기관은 섹션에 자세히 설명된 메커니즘을 사용하여 진행 정보를 생성합니다. 3.5.1을 사용하여 데이터 흐름 그래프에 전파합니다. 정보는 일반적으로 시스템에서 가장 오래되고 처리되지 않은 기록을 반영하며 순서가 잘못된 기록을 승인 및/또는 처리하기 위한 자연 한계를 설정합니다.

순차 시스템과 달리 기록은 자연 한도를 초과하지 않는 한 도착 순서에 따라 자체 없이 처리됩니다. 순차 시스템이 무질서를 허용하는 경우에도 이를 운영자 기반으로 적용하는데, 이는 시스템 전체의 진행 상황 추적 메커니즘 없이는 비효율적이고 불확실합니다.

### 3.4 장애의 영향

무한한 데이터 처리에서 무질서는 진행을 방해하거나 [108] 무시할 경우 잘못된 결과를 초래할 수 있습니다 [142].

장애는 계산 토플로지를 구성하는 연산자가 요구할 때 처리 진행에 영향을 미칩니다.

Sect. 3.5. Finally, in Sect. 3.6, we discuss the differences of out-of-order data management in early and modern systems and we present open problems in Sect. 3.7.

### 3.1 Causes of disorder

Disorder in data streams may be due to stochastic factors that are external to a streaming system or to the operations taking place inside the system.

The most common external factor that introduces disorder to streams is the network [99, 142]. Depending on the network's reliability, bandwidth, and load, the routing of some stream records can take longer to complete compared to the routing of others, leading to a different arrival order in terms of event time in a streaming system. Even if the order of records in an individual stream is preserved, ingestion from multiple sources, such as sensors, even with synchronized clocks, typically results in a disordered collection of records. Notably, if the sources do not feature synchronized clocks, which is often the case, the generated timestamps of records may not correspond to the real time of production. Consequently, if the timestamps used to specify order do not represent event time, this cause of disorder is impossible to fix.

External factors aside, specific operations on streams break record order. First, join processing takes two streams and produces a shuffled combination of the two, since a parallel join operator repartitions the data according to the join attribute [157] and outputs join results by order of match [77, 90]. Second, windowing based on an attribute different to the ordering attribute reorders the stream [54]. Third, data prioritization [130, 156] by using an attribute different to the ordering one also changes the stream's order. Finally, the union operation on two unsynchronized streams yields a stream with all records of the two input streams interleaving each other in random order [12].

### 3.2 Disorder and processing progress

In order to manage disorder, streaming systems need to detect and measure processing progress. *Progress* regards how much the processing of a stream's records has advanced over time. Processing progress can be defined and quantified with the aid of an attribute  $A$  of a stream's records that orders the stream. The processing of the stream progresses when the smallest value of  $A$  among the unprocessed records increases over time [108]. A then is a *progressing attribute* and the smallest (unprocessed) value of  $A$  per se, is a measure of progress because it denotes how far in the processing of records the system has reached since the beginning. Processing progress can be quantified using more than one attribute of a stream's records.

A streaming system's capacity to track processing progress enables the system to decide a *lateness bound* in order to quantify the level of disorder that a streaming system can accept while it makes processing progress. To elaborate, lateness bound is a measure of how late, with respect to the rest of the stream, data can be accepted for processing by a streaming system. It typically corresponds to a time measure or count of stream records. For example, a lateness bound of 1 s makes a streaming system's window operator to compute the window's function 1 s later than the window's time span. The time extension allows the inclusion of out-of-order records to the window computation.

### 3.3 System architectures for managing disorder

Two main architectural archetypes have influenced the design of streaming systems with respect to managing disorder: (i) in-order processing systems [12, 21, 54, 142], and (ii) out-of-order processing systems [14, 37, 108, 120].

In-order processing systems manage disorder using one of three main strategies. First, they can assume input streams are ordered and discard late data [12, 21, 54]. Second, they can buffer and reorder input streams up to a lateness bound to enforce stream order [12, 21, 142]. Then, they forward the reordered records for processing and clear the corresponding buffers. Finally, they can admit late data without reordering up to a lateness bound [12]. Although the last strategy supports disordered input, it entails that downstream operators should also quantify and enforce a lateness bound independently. In order to do so effectively, they need to track processing progress, which is hard to achieve without a system-wide progress tracking mechanism. Therefore, in-order systems commonly enforce and preserve stream order and use it to deduce processing progress.

In out-of-order processing systems, operators or a global authority produce progress information using any of the mechanisms detailed in Sect. 3.5.1, and propagate it to the dataflow graph. The information typically reflects the oldest unprocessed record in the system and establishes a lateness bound for admitting and/or processing out-of-order records. In contrast to in-order systems, records are processed without delay in their arrival order, as long as they do not exceed the lateness bound. Even when in-order systems allow disorder, they apply it on an operator basis, which is inefficient and uncertain without a system-wide progress-tracking mechanism.

### 3.4 Effects of disorder

In unbounded data processing, disorder can impede progress [108] or lead to wrong results if ignored [142].

Disorder affects processing progress when the operators that comprise the topology of the computation require

주문된 입력. Join 및 Aggregate의 다양한 구현은 올바른 결과를 생성하기 위해 정렬된 입력에 의존합니다 [12, 142]. 순차 시스템의 운영자는 비순차적 레코드를 수신하면 일반적으로 자신이 속한 창에 포함하기 전에 순서를 변경합니다. 그러나 재정렬에는 처리 오버헤드, 메모리 공간 오버헤드 및 대기 시간이 발생합니다.

반면에 비순차적 시스템은 진행 상황을 추적하고 도착 순서에 관계없이 자연 한계까지 데이터를 처리하며 자연 한계 이전의 데이터와 관련된 처리 상태를 제거합니다. 참고로, Apply, Project, Select, Dupelim 및 Union과 같은 순서를 구분하지 않는 연산자 [12, 108, 142]는 스트림의 무질서에 불가지론적이며 무질서한 입력이 표시되는 경우에도 올바른 결과를 생성합니다.

잘못된 순서의 데이터를 무시하면 입력 데이터의 불완전한 하위 집합에서 출력이 계산되므로 많은 사용 사례에서 잘못된 결과가 발생할 수 있습니다. 따라서 스트리밍 시스템은 비순차적 데이터를 처리하고 그 효과를 계산에 통합할 수 있어야 합니다. 데이터 장애를 수용하는 것은 차단 연산자의 처리 차단을 해제하고 해당 계산 상태를 제거하는 데 중요합니다. 다음으로 장애 관리 메커니즘을 통해 이를 달성할 수 있는 방법에 대해 논의합니다.

### 3.5 장애 관리 메커니즘

이 섹션에서는 제한되지 않은 데이터의 장애를 관리하기 위한 영향력 있는 메커니즘, 즉 슬랙 [12], 하트비트 [142], 로우 워터마크 [108], 포인트 스템프 [120] 및 트리거 [16]에 대해 자세히 설명합니다. 이러한 메커니즘은 시간과 같은 지표를 사용하여 자연 한계를 수량화하고 스트리밍 시스템에서 처리 진행 상황을 추적하는 데 활용됩니다. 자연 제한이 만료된 후에 레코드가 도착하면 트리거를 사용하여 개정 처리 시 계산 결과를 업데이트할 수 있습니다 [11].

우리는 또한 장애 관리의 수단으로 많이 사용되어 온 데이터 흐름 그래프 전반에 걸쳐 정보를 전달하기 위한 일반적인 메커니즘인 구두점 [155]에 대해 논의합니다. 구두점은 데이터에 포함된 메타데이터 주석입니다.

스트림. 구두점 자체는 스트림 데이터 레코드의 속성을 식별하는 일련의 패턴으로 구성된 스트림 레코드입니다.

#### 3.5.1 진행 상황 추적 메커니즘

우리는 슬랙, 하트비트, 로우 워터마크, 포인트 스템프를 자세히 설명하고 묘사합니다. 그림 2는 슬랙, 하트비트, 로우 워터마크 간의 차이점을 보여줍니다. 그림은 t=1에서 시작하는 4초 이벤트 시간 텀블링 창에서 레코드를 계산하는 간단한 단계 연산자를 보여줍니다. 연산자는 이벤트 시간이 창의 종료 타임스탬프를 지나서 진행되었다는 표시를 기다리므로 창당 단계를 계산하고 출력합니다. 표시는 진행률 추적 메커니즘에 따라 다릅니다. 이 연산자에 대한 입력은 타임스탬프만 포함하는 7개의 레코드입니다.

t=1에서 t=7까지. 타임스탬프는 레코드가 입력 소스에서 생성된 이벤트 시간(초)을 나타냅니다.

각 레코드에는 서로 다른 타임스탬프가 포함되어 있으며 모든 레코드는 타임스탬프의 오름차순으로 소스에서 전달됩니다. 네트워크 지연으로 인해 레코드가 순서대로 스트리밍 시스템에 도착할 수 있습니다.

3.5.1.1 Slack은 특정 측정항목의 고정된 양에 대해 순서가 잘못된 데이터를 기다리는 간단한 메커니즘입니다.

Slack은 원래 순서가 잘못된 레코드의 실제 발생과

시간에 맞춰 도착했다면 입력 스트림에서 가질 위치입니다. 그러나 경과 시간을 기준으로 정량화할 수도 있습니다. 기본적으로 Slack은 늦은 기록에 대해 고정된 유예 기간을 표시합니다.

그림 2a는 슬랙 메커니즘을 보여줍니다. 비순차적 레코드를 수용하기 위해 운영자는 slack=1 까지 비순차적 레코드를 허용합니다. 따라서 t=1 및 t=2(그림에는 표시되지 않음)의 레코드를 허용한 운영자는 t=4의 레코드를 수신하게 됩니다. 레코드의 타임스탬프는 간격 [0, 4]에 대한 첫 번째 창의 최대 타임스탬프와 일치합니다. 일반적으로 이 레코드로 인해 운영자는 창을 닫고 단계를 계산 및 출력하게 되지만, 여유 값으로 인해 운영자는 레코드를 하나 더 수신할 때까지 기다립니다. t=3인 다음 레코드는 첫 번째 창에 속하며 거기에 포함됩니다. 이 시점에서 Slack도 앞으로 이동하고 이 이벤트는 최종적으로 창 계산을 트리거하여 t=[1, 2, 3]에 대해 C=3을 출력합니다. 대조적으로, 연산자는 입력의 고리 부분에서 t=5를 허용하지 않습니다. 그 이유는 입력이 자연 순서 이후에 두 개의 레코드에 도착하고 여유 값에 포함되지 않기 때문입니다.

#### 3.5.1.2 하트비트는 다음으로 구성된 Slack의 대안입니다.

데이터 스트림에 대한 진행 정보를 전달하는 외부 신호. 여기에는 모든 후속 스트림 레코드가 하트비트의 타임스탬프보다 큰 타임스탬프를 갖게 될을 나타내는 타임스탬프가 포함되어 있습니다. 하트비트는 입력 소스에 의해 생성되거나 네트워크 대기 시간 제한, 입력 소스 간 애플리케이션 클릭 스큐, 잘못된 데이터 생성과 같은 환경 매개변수를 관찰하여 시스템에서 추론할 수 있습니다 [142].

그림 2b는 하트비트 메커니즘을 보여줍니다. 입력 관리자는 들어오는 레코드를 타임스탬프별로 버퍼링하고 정렬합니다.

버퍼링된 레코드 수(이 예에서는 2개(t = 5, t = 6))는 중요하지 않습니다. 소스는 주기적으로 입력 관리자에게 하트비트, 즉 타임스탬프가 있는 신호를 보냅니다.

그런 다음 입력 관리자는 타임스탬프가 하트비트의 타임스탬프보다 작거나 같은 모든 레코드를 오름차순으로 운영자에게 전달합니다. 하트비트는 입력 스트림 외부에 있지만 그림에서는 이벤트 순서를 표시하기 위해 입력 레코드 옆에 하트비트(h = 2, h = 4)를 배치합니다. 예를 들어, 타임스탬프 t = 2를 전달하는 하트비트 h = 2가 입력 관리자(그림에 표시되지 않음)에 도착하면 입력 관리자는 타임스탬프 t = 2가 있는 레코드를 디스패치합니다.

ordered input. Various implementations of *join* and *aggregate* rely on ordered input to produce correct results [12, 142]. When operators in in-order systems receive out-of-order records, they typically reorder them prior to including them in the window they belong. Reordering, however, imposes processing overhead, memory space overhead, and latency. Out-of-order systems, on the other hand, track progress and process data in whatever order they arrive, up to the lateness bound, and remove processing state related to data earlier than the lateness bound. As a sidenote, order-insensitive operators [12, 108, 142], such as *apply*, *project*, *select*, *dupelim*, and *union*, are agnostic to disorder in a stream and produce correct results even when presented with disordered input.

Ignoring out-of-order data can lead to incorrect results in many use cases, since the output is computed on an incomplete subset of the input data. Thus, a streaming system needs to be capable of processing out-of-order data and incorporate their effect to the computation. Embracing data disorder is important for unblocking the processing of blocking operators and purging corresponding computation state. We next discuss how it can be achieved with disorder management mechanisms.

### 3.5 Mechanisms for managing disorder

In this section, we elaborate on influential mechanisms for managing disorder in unbounded data, namely slack [12], heartbeats [142], low-watermarks [108], pointstamps [120], and triggers [16]. These mechanisms quantify a lateness bound using a metric, such as time, and are leveraged by streaming systems to track processing progress. If records arrive after the lateness bound expires, triggers can be used to update computation results in *revision processing* [11].

We also discuss punctuations [155], a generic mechanism for communicating information across the dataflow graph, that has been heavily used as a vehicle in managing disorder. Punctuations are metadata annotations embedded in data streams. A punctuation is itself a stream record, which consists of a set of patterns each identifying an attribute of a stream data record.

#### 3.5.1 Progress tracking mechanisms

We detail and depict slack, heartbeats, low-watermark, and pointstamps. Figure 2 showcases the differences between slack, heartbeats, and low-watermarks. The figure depicts a simple aggregation operator that counts records in 4-second event-time tumbling windows starting at  $t=1$ . The operator awaits for some indication that event time has advanced past the end timestamp of a window, so that it computes and outputs an aggregate per window. The indication varies according to the progress-tracking mechanism. The input to this operator are seven records containing only a timestamp

from  $t=1$  to  $t=7$ . The timestamp signifies the event time in seconds that the record was produced in the input source. Each record contains a different timestamp and all records are dispatched from a source in ascending order of timestamp. Due to network latency, the records may arrive to the streaming system out of order.

**3.5.1.1 Slack** is a simple mechanism that involves waiting for out-of-order data for a fixed amount of a certain metric. Slack originally denoted the number of records intervening between the actual occurrence of an out-of-order record and the position it would have in the input stream if it arrived on time. However, it can also be quantified in terms of elapsed time. Essentially, slack marks a fixed grace period for late records.

Figure 2a presents the slack mechanism. In order to accommodate out-of-order records, the operator admits out-of-order records up to  $slack=1$ . Thus, the operator, having admitted records with  $t=1$  and  $t=2$  (not depicted in the figure), will receive the record with  $t=4$ . The timestamp of the record coincides with the maximum timestamp of the first window for interval  $[0, 4]$ . Normally, this record would cause the operator to close the window and compute and output the aggregate, but because of the slack value, the operator will wait to receive one more record. The next record with  $t=3$  belongs to the first window and is included there. At this point, slack also moves forward and this event finally triggers the window computation, which outputs  $C=3$  for  $t=[1, 2, 3]$ . In contrast, the operator will not accept  $t=5$  at the tail of the input, because it arrives two records after its natural order and is not covered by the slack value.

**3.5.1.2 A heartbeat** is an alternative to slack that consists of an external signal carrying progress information about a data stream. It contains a timestamp indicating that all succeeding stream records will have a timestamp larger than the heartbeat's timestamp. Heartbeats can either be generated by an input source or deduced by the system by observing environment parameters, such as network latency bound, application clock skew between input sources, and out-of-order data generation [142].

Figure 2b depicts the heartbeat mechanism. An input manager buffers and orders the incoming records by timestamp. The number of records buffered, two in this example ( $t = 5, t = 6$ ), is of no importance. The source periodically sends a heartbeat to the input manager, i.e. a signal with a timestamp. Then, the input manager dispatches to the operator all records with timestamp less or equal to the timestamp of the heartbeat in ascending order. Although heartbeats are external to the input stream, in the Figure we position heartbeats ( $h = 2, h = 4$ ) alongside input records just to show the order of events. For instance, when the heartbeat  $h = 2$ , which carries a timestamp  $t = 2$ , arrives in the input manager (not shown in the figure), the input manager dispatches the records with timestamp  $t =$

## 스트림의 진화에 관한 조사…

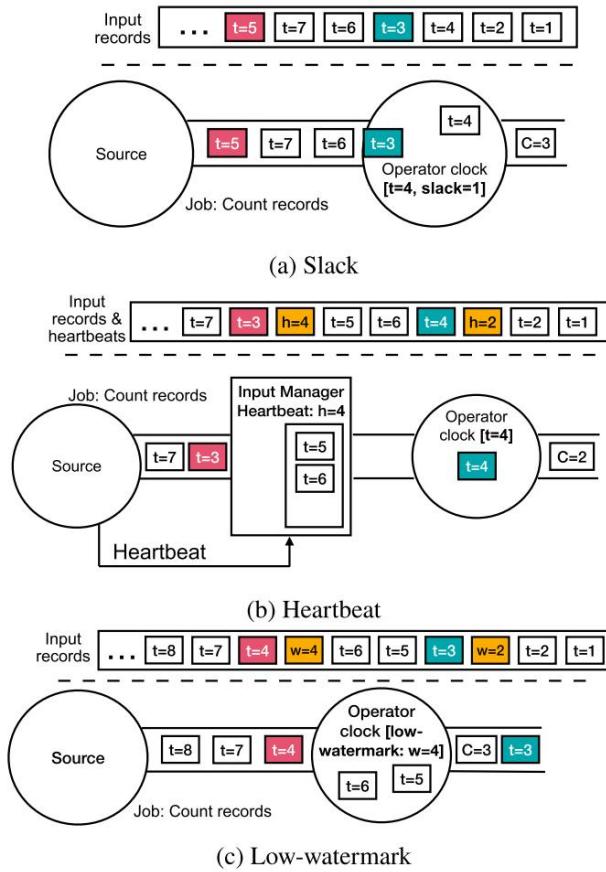


그림 2 장애 관리 메커니즘

1 및  $t = 2$ 입니다. 그런 다음 입력 관리자는  $t = 4, t = 6, t = 5$ 인 레코드를 이 순서대로 수신하여 올바른 순서로 배치합니다. 타임스탬프  $t = 4$ 를 전달하는 하트비트  $h = 4$ 가 도착하면 입력 관리자는 타임스탬프  $t = 4$ 가 있는 레코드를 운영자에게 전달합니다. 이 레코드는 간격  $[0, 4]$ 에 대한 첫 번째 창 계산을 트리거합니다.

연산자는 그림에 표시되지 않은  $t = [1, 2]$ 인 두 개의 레코드를 세어  $C = 2$ 를 출력합니다. 입력 관리자는 타임스탬프  $t = 4$ 인 최신 하트비트보다 오래되었으므로 타임스탬프  $t = 3$ 인 수신 레코드를 무시합니다.

### 3.5.1.3 스트림의 속성 A에 대한 하위 수위는 스트림의 특정 하위 집합 내에서 A의 가장 낮은 값입니다.

따라서 향후 레코드는 일반적으로 동일한 속성에 대한 현재 최저 수위보다 높은 값을 갖게 됩니다. A는 레코드의 이벤트 타임스탬프인 경우가 많습니다. 이 메커니즘은 스트리밍 시스템에서 A의 하위 워터마크를 통해 처리 진행 상황을 추적하고 속성 A의 값이 하위 워터마크보다 작지 않은 순서가 잘못된 데이터를 허용하는 데 사용됩니다. 또한 스트리밍 조인 계산의 해당 해시 테이블 항목과 같이 A에 대해 유지되는 상태를 제거하는 데 사용될 수 있습니다. 또한 차단 해제를 위한 스트림 진행 상황을 정확하게 예측하기 위해 워터마크도 활용되었습니다.

최적화된 스케줄링을 통해 처리량 및 대기 시간 성능을 향상시키는 창 계산.

그림 2c는 시스템에서 가장 오래 보류 중인 작업을 나타내는 로우 워터마크 메커니즘을 나타냅니다. 여기에서 낮은 워터마크 타임스탬프를 포함하는 구두점은 창이 닫히고 계산되는 시기를 결정합니다.  $t = 1$  및  $t = 2$ 인 두 개의 레코드,  $t = 2$ (다운스트림으로 전파됨)에 해당하는 낮은 워터마크 및 레코드  $t = 3$ 을 수신한 후 운영자는 레코드  $t = 5$ 를 수신합니다. 첫 번째 창의 종료 타임스탬프인 4보다 크거나 같은 이벤트 타임스탬프는 창이 실행되거나 닫히는 원인일 수 있습니다. 그러나 이 접근 방식은 순서가 잘못된 데이터를 설명하지 않습니다. 대신, 연산자가  $t = 4$ 인 하위 워터마크를 수신하면 창이 닫힙니다. 이 시점에서 연산자는  $t = [1, 2, 3]$ 에 대해  $C = 3$ 을 계산하고  $t = [5, 6]$ 인 레코드를 다음 항목에 할당합니다. 간격  $[4, 8)$ 의 두 번째 창. 운영자는 현재 로우 워터마크 값  $t = 4$ 보다 크지 않기 때문에(더 최근의) 레코드  $t = 4$ 를 인정하지 않습니다.

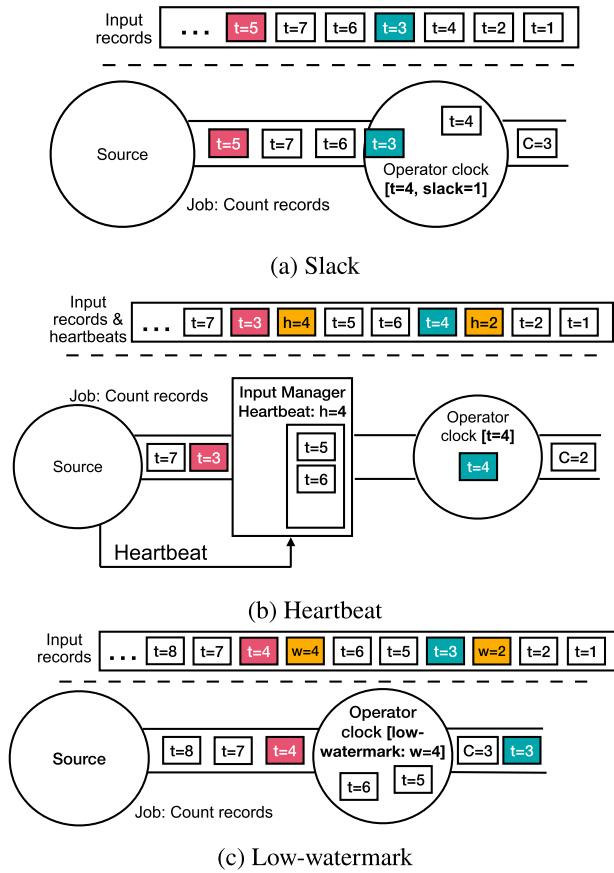
### 3.5.1.4 하트비트, 슬랙, 로우 워터마크, 구두점 비교. 하트비트와 여유 시간은 모두 데이터 스트림 외부에 있습니다. 하트비트는 입력 소스에서 스트리밍 시스템의 수집 지점으로 전달되는 신호입니다. 사용자에게 숨겨진 스트리밍 시스템의 내부 메커니즘인 하트비트와 달리, 슬랙은 사용자가 제공하는 쿼리 사양의 일부이다 [12].

하트비트와 로우 워터마크는 진행률 추적 논리 측면에서 유사합니다. 그러나 한 가지 중요한 차이점이 이들을 구별합니다. 로우 워터마크는 타임스탬프뿐만 아니라 스트림 레코드의 모든 진행 속성에 대해 현재 진행 지점을 나타내는 가장 오래된 값의 개념을 일반화합니다.

하트비트 및 여유 시간과 달리 구두점은 연산자가 생성한 레코드 속성의 하위 워터마크 [108], 이벤트 시간 외곽 [142] 또는 여유 시간 [12]과 같은 진행 정보를 전달하기 위한 채널을 제공합니다. 따라서 구두점은 입력 스트림에 더 이상 표시되지 않는 데이터를 전달할 수 있습니다. 예를 들어 데이터는 특정 값보다 작은 타임스탬프로 기록됩니다. 구두점은 상태 관리, 모니터링, 흐름 제어 등 스트리밍 시스템의 다른 기능 영역에서도 유용합니다.

### 3.5.1.5 구두점과 같은 포인트스탬프는 데이터 스트림에 포함되지만 별도의 레코드를 형성하는 구두점과 달리 포인트스탬프는 각 스트림 데이터 레코드에 첨부됩니다. 포인트스탬프는 특정 시점에 데이터 흐름 그래프의 정점이나 가장자리에 데이터 레코드를 배치하는 타임스탬프와 위치의 쌍입니다. 타임스탬프 $t$ 가 있는 위치 $l$ 의 처리되지 않은 레코드 $p$ 는 $p$ 가 다음과 같은 경우 타임스탬프 $t'$ 가 있는 위치 $l$ 의 처리되지 않은 또 다른 레코드 $p'$ 가 될 수 있습니다.

$|$  위치에 도착할 수 있다       $t'$  타임스탬프  $t'$  이전 또는 그 시점에 . 타임스탬프  $t$ 가 있는 위치  $l$ 의 처리되지 않은 레코드  $p$ 는 처리되지 않은 다른 레코드가 없을 때 처리 진행의 최전선에 있습니다.



**Fig. 2** Mechanisms for managing disorder

1 and  $t = 2$  to the operator. The input manager then receives records with  $t = 4$ ,  $t = 6$ , and  $t = 5$  in this order and puts them in the right order. When the heartbeat  $h = 4$ , which carries a timestamp  $t = 4$ , arrives, the input manager dispatches the record with timestamp  $t = 4$  to the operator. This record triggers the computation of the first window for interval  $[0, 4]$ . The operator outputs  $C = 2$  counting two records with  $t = [1, 2]$  not depicted in the figure. The input manager ignores the incoming record with timestamp  $t = 3$ , as it is older than the latest heartbeat with timestamp  $t = 4$ .

**3.5.1.3 The low-watermark** for an attribute  $A$  of a stream is the lowest value of  $A$  within a certain subset of the stream. Thus, future records will typically bear a higher value than the current low-watermark for the same attribute. Often,  $A$  is a record's event time timestamp. The mechanism is used by a streaming system to track processing progress via the low-watermark for  $A$ , to admit out-of-order data whose attribute  $A$ 's value is not smaller than the low-watermark. Further, it can be used to remove state that is maintained for  $A$ , such as the corresponding hash table entries of a streaming join computation. In addition, watermarks have also been leveraged to accurately estimate stream progress for unblocking

windowed computations that lead to throughput and latency performance improvements via optimized scheduling [63].

Figure 2c presents the low-watermark mechanism, which signifies the oldest pending work in the system. Here, punctuations carrying the low-watermark timestamp decide when windows will be closed and computed. After receiving two records with  $t = 1$  and  $t = 2$ , the corresponding low-watermark for  $t = 2$  (which is propagated downstream), and record  $t = 3$ , the operator receives record  $t = 5$ . Since this record carries an event time timestamp greater or equal to 4, which is the end timestamp of the first window, it could be the one to cause the window to fire or close. However, this approach would not account for out-of-order data. Instead, the window closes when the operator receives the low-watermark with  $t = 4$ . At this point, the operator computes  $C = 3$  for  $t = [1, 2, 3]$  and assigns records with  $t = [5, 6]$  to the second window with interval  $[4, 8]$ . The operator will not admit record  $t = 4$  because it is not greater (more recent) than the current low-watermark value  $t = 4$ .

**3.5.1.4 Comparison among heartbeats, slack, low-watermark, and punctuations.** Heartbeats and slack are both external to a data stream. Heartbeats are signals communicated from an input source to a streaming system's ingestion point. Differently from heartbeats, which are an internal mechanism of a streaming system hidden from users, slack is part of the query specification provided by users [12].

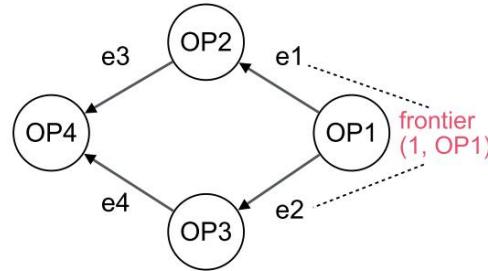
Heartbeats and low-watermarks are similar in terms of progress-tracking logic. However, one important difference sets them apart. The low-watermark generalizes the concept of the oldest value, which signifies the current progress point, to any progressing attribute of a stream record, not just timestamps.

In contrast to heartbeats and slack, *punctuations* provide a channel for communicating progress information such as a record attribute's low-watermark produced by an operator [108], event time skew [142], or slack [12]. Thus, punctuations can convey which data no longer appear in an input stream; for instance, the data records with smaller timestamps than a specific value. Punctuations are useful in other functional areas of a streaming system as well, such as state management, monitoring, and flow control.

**3.5.1.5 Pointstamps**, like punctuations, are embedded in data streams, but a pointstamp is attached to each stream data record as opposed to a punctuation, which forms a separate record. A pointstamp, is a pair of timestamp and location that positions data records on a vertex or edge of the dataflow graph at a specific point in time. An unprocessed record  $p$  at location  $l$  with timestamp  $t$  could result in another unprocessed record  $p'$  at location  $l'$  with timestamp  $t'$  when  $p$  can arrive at location  $l'$  before or at timestamp  $t'$ . An unprocessed record  $p$  at location  $l$  with timestamp  $t$  is in the *frontier* of processing progress when no other unprocessed records

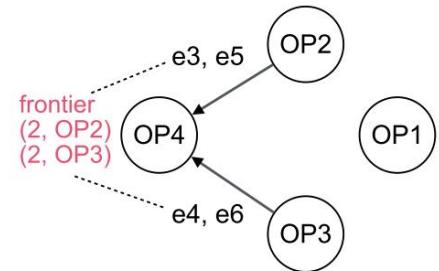
그림 3 포인트 스템프와 프론티어의 높은 수준의 작업 흐름

Active Pointstamp	Unprocessed Event(s)	Occurrence Count	Precursor Count
(1, OP1)	e1, e2	2	0
(2, OP2)	e3	1	1 (1, OP1)
(2, OP3)	e4	1	1 (1, OP1)



a) Pointstamps and frontier

Active Pointstamp	Unprocessed Event(s)	Occurrence Count	Precursor Count
(2, OP2)	e3, e5	2	0
(2, OP3)	e4, e6	2	0



b) Frontier moves forward

p.가 발생할 수 있습니다. 따라서 앞서 언급한 레코드 p가 처리되면 프론티어는 계속 이동합니다. 시스템은 향후 레코드가 해당 레코드를 생성한 레코드보다 더 큰 타임스탬프를 갖도록 강제합니다. 처리 진행 상황에 대한 이러한 모델링은 타임스탬프를 사용하여 데이터 흐름 그래프에서 데이터 기록 과정을 추적하고 현재 경계선을 계산하기 위해 처리되지 않은 이벤트 간의 종속성을 추적합니다.

이러한 관점에서 프론티어의 기능은 로우 워터마크와 유사합니다.

그림 3에 표시된 예는 포인트 스템프와 프론티어가 작동하는 방식을 보여줍니다. 그림 3a의 예에는 3개의 활성 포인트 스템프가 포함되어 있습니다. 포인트 스템프는 하나 이상의 처리되지 않은 이벤트에 해당할 때 활성화됩니다. 포인트 스템프 (1, OP1)는 포인트 스템프(1, OP1)로 이어질 수 있는 활성 포인트 스템프가 없기 때문에 처리 진행의 최전선에 있습니다. 다른 포인트 스템프가 발생할 수 있는 포인트 스템프 수는 전구체 수에 따라 지정됩니다. 결과적으로 포인트 스템프(1, OP1)의 전구체 수는 0입니다.

프론티어에서는 처리되지 않은 이벤트에 대한 알림을 전달할 수 있습니다. 따라서 처리되지 않은 이벤트 e1과 e2는 각각 OP2와 OP3에 전달될 수 있습니다. 이벤트 e1과 e2 모두 동일한 포인트 스템프를 갖기 때문에 발생 횟수는 2입니다.

데이터 흐름 그래프의 이 스냅샷을 보면 포인트 스템프 (1, OP1)가 포인트 스템프 (2, OP2) 및 (2, OP3)로 이어질 수 있음을 쉽게 알 수 있습니다. 따라서 후자의 두 포인트 스템프 각각의 전구체 수는 1입니다.

조금 후에 그림 3b에서 설명한 것처럼 이벤트 e1과 e2가 각각 OP2와 OP3에 전달됩니다. 처리 결과 각각 처리되지 않은 이벤트 e3 및 e4와 동일한 포인트 스템프를 갖는 새로운 이벤트 e5 및 e6이 생성됩니다.

타임스탬프 1이 있는 처리되지 않은 이벤트가 더 이상 없고 포인트 스템프(2, OP2) 및 (2, OP3)의 전구체 수가 0이므로 프론티어는 이러한 활성 포인트 스템프로 이동합니다.

결과적으로 네 가지 이벤트 알림이 모두 전달될 수 있습니다.

사용되지 않는 포인트 스템프(1, OP1)가 해당 위치에서 제거되었습니다.

처리되지 않은 이벤트에 해당하지 않기 때문입니다. 이벤트 e3, e4, e5 및 e6이 전달된 후 포인트 스템프 (2, OP2) 및 (2, OP3)에도 동일한 일이 발생합니다. 이 예제는 데모 목적으로 간단하게 만들어졌지만 진행률 추적 메커니즘에는 임의의 반복 및 중첩 계산의 진행률을 추적할 수 있는 기능이 있습니다.

### 3.5.2 순환 쿼리에서 순서가 잘못된 데이터의 진행 상황 추적

순환 쿼리는 진행 상황을 추적하기 위해 특별한 처리가 필요합니다.

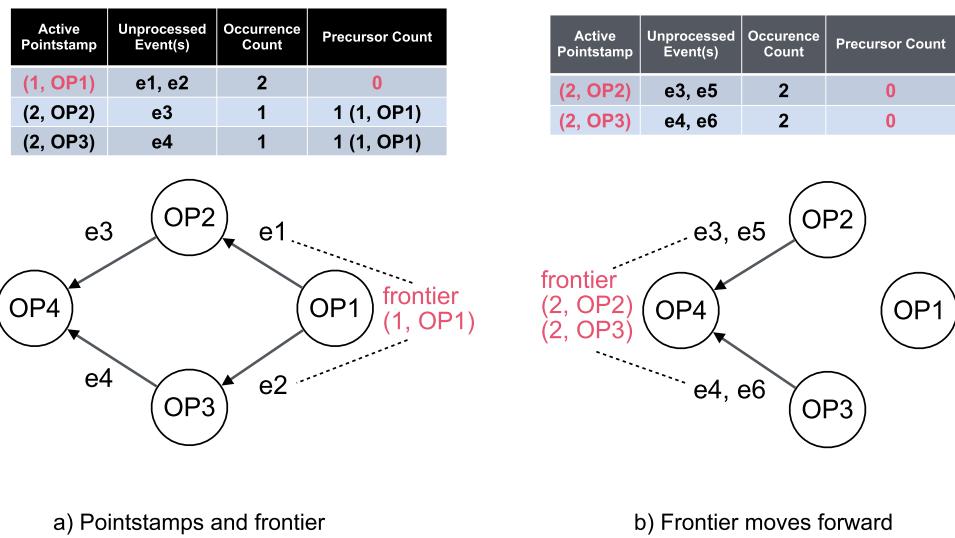
순환 쿼리에는 항상 조인이나 유니온과 같은 이항 연산자가 포함됩니다. 이항 연산자에 의해 생성된 출력은 이항 연산자의 입력 채널 중 하나에 다시 연결되는 데이터 흐름 그래프에서 추가 루프를 만납니다. 예를 들어 구두점을 사용하는 진행 모델에서 이항 연산자는 두 입력 채널 모두에 구두점이 나타날 때만 구두점을 전달하고, 그렇지 않으면 둘 다 도착할 때까지 기다리지 않습니다.

이항 연산자의 입력 채널 중 하나가 자체 출력 채널에 의존하므로 자연이 불가피합니다.

Chandramouli et al. [46]은 순환 스트리밍 쿼리의 진행 상황을 즉석에서 감지하기 위한 연산자를 제안합니다. 연산자는 전달되는 이벤트의 타임스탬프에서 발생된 추론적 구두점을 루프에 도입합니다. 구두점이 루프에 흐르는 동안 연산자는 스트림의 레코드를 관찰하여 추측을 검증합니다. 추측성 구두점이 연산자에 다시 입력되어 유효성이 검사되면 진행 정보를 다운스트리밍으로 전달하는 일반 구두점이 됩니다. 그런 다음 새로운 추측 구두점이 생성되어 루프에 입력됩니다. 전용 연산자, 예측 출력 및 구두점을 결합함으로써 이 작업은 진행 상황을 추적하고 순환 스트리밍 쿼리의 장애를 허용할 수 있습니다.

이 접근 방식에서는 루프가 a) 올바른 추측을 할 수 있고, b) 앞으로 나아갈 수 있으며, c) 구두점으로 인해 차단되지 않는 연산자로 구성됩니다. 작동-

**Fig. 3** High-level workflow of pointstamps and frontier



could result in  $p$ . Thus, when the aforementioned record  $p$  is processed, the frontier moves on. The system enforces that future records will bear a greater timestamp than the records that generated them. This modeling of processing progress traces the course of data records on the dataflow graph with timestamps and tracks the dependencies between unprocessed events in order to compute the current frontier. Under this light, the function of a frontier is similar to a low-watermark.

The example shown in Fig. 3 showcases how pointstamps and frontiers work. The example in Fig. 3a includes three active pointstamps. Pointstamps are active when they correspond to one or more unprocessed events. Pointstamp (1, OP1) is in the frontier of processing progress, because there are no active pointstamps that could result in pointstamp (1, OP1). The number of pointstamps that can result in another pointstamp are specified by the precursor count. Consequently, the precursor count of pointstamp (1, OP1) is zero.

In the frontier, notifications for unprocessed events can be delivered. Accordingly, unprocessed events e1 and e2 can be delivered to OP2 and OP3 respectively. The occurrence count is 2 because both events e1 and e2 bear the same pointstamp. Looking at this snapshot of the dataflow graph, it is easy to see that pointstamp (1, OP1) could result in pointstamps (2, OP2) and (2, OP3). Therefore, the precursor count of each of the latter two pointstamps is 1.

A bit later, as Fig. 3b depicts, events e1 and e2 are delivered to OP2 and OP3, respectively. Their processing results in the generation of new events e5 and e6, which bear the same pointstamp as unprocessed events e3 and e4, respectively. Since there are no more unprocessed events with timestamp 1, and the precursor counts of pointstamps (2, OP2) and (2, OP3) are 0, the frontier moves on to these active pointstamps. Consequently, all four event notifications can be delivered. The obsolete pointstamp (1, OP1) is removed from its loca-

tion, since it corresponds to no unprocessed events. The same will happen to pointstamps (2, OP2) and (2, OP3), following the delivery of events e3, e4, e5, and e6. Although this example is made simple for demonstration purposes, the progress tracking mechanism has the power to track the progress of arbitrary iterative and nested computations.

### 3.5.2 Tracking progress of out-of-order data in cyclic queries

Cyclic queries require special treatment for tracking progress. A cyclic query always contains a binary operator, such as a join or a union. The output produced by the binary operator meets a loop further in the dataflow graph that connects back to one of the binary operator's input channels. In a progress model that uses punctuations for instance, the binary operator forwards a punctuation only when it appears in both of its input channels, otherwise it blocks waiting for both to arrive. Since one of the binary operator's input channels depends on its own output channel, a stall is inevitable.

Chandramouli et al. [46] propose an operator for detecting progress in cyclic streaming queries on the fly. The operator introduces a speculative punctuation in the loop that is derived from the passing events' timestamps. While the punctuation flows in the loop, the operator observes the stream's records to validate its guess. When the speculative punctuation re-enters the operator and is validated, it becomes a regular punctuation that carries progress information downstream. Then, a new speculative punctuation is generated and is fed to the loop. By combining a dedicated operator, speculative output, and punctuations, this work is able to track progress and tolerate disorder in cyclic streaming queries. The approach entails that a loop consists of operators that a) are able to make correct speculations, b) make forward progress, and c) will not block due to a punctuation. Oper-

## 스트림의 진화에 관한 조사…

루프의 작업자는 원래의 추측 구두점을 수정할 수 있습니다. 이 접근 방식은 강력하게 수렴되는 쿼리에 대해 추론적 출력을 제공하는 시스템에 적용될 수 있으며, 이는 각 결과에 대한 유한 파생과 함께 유한 입력에 대한 유한 결과를 제공합니다.

Naiad [120, 121]에서 일반적인 진행 추적 모델은 논리적, 즉 처리 시간, 이벤트에 첨부된 다차원 타임 스템프를 특징으로 합니다. 각 타임스탬프는 이벤트가 속한 입력 배치와 이벤트가 통과하는 각 루프에 대한 반복 카운터로 구성됩니다. Chandramouli et al. [46], Naiad는 특수 연산자를 활용하여 순환 쿼리를 지원합니다. 그러나 연산자는 루프에 들어가는 이벤트의 반복 카운터를 증가시키는 데 사용됩니다. 진행을 보장하기 위해 시스템에서는 이벤트 핸들러가 현재 처리 중인 이벤트의 타임스탬프보다 더 큰 타임스탬프를 가진 메시지만 전달하도록 허용합니다. 이 제한은 보류 중인 모든 이벤트에 대해 부분 순서를 적용합니다. 순서는 출력 생성에 대한 알림을 전달하기 위해 이벤트 처리 완료의 가장 빠른 논리적 시간을 계산하는 데 사용됩니다. Naiad의 진행 상황 추적 메커니즘은 a) 데이터 흐름 노드에 효율적인 알림 전달을 제공하여 확장성을 제공하고 b) 중복 작업을 방지하는 충분 계산을 가능하게 합니다.

### 3.5.3 개정 처리

개정 처리는 지역, 업데이트 또는 철회된 데이터에 대한 계산 업데이트로, 올바른 결과를 제공하기 위해 이전 출력을 수정해야 합니다.

개정 처리는 Borealis [11]에서 데뷔했습니다. 이후부터 순차 처리 아키텍처 [45, 122] 및 비순차 처리 아키텍처 [16, 17, 31, 99]와 결합되었습니다. 일부 접근 방식에서 개정 처리는 수신 데이터를 저장하고 지역, 업데이트 또는 철회된 데이터에 직면하여 계산을 수정하는 방식으로 작동합니다 [16, 17, 31]. 다른 접근 방식은 영향을 받은 데이터를 재생하고, 계산을 수정하고, 수정 메시지를 전파하여 영향을 받은 모든 결과를 현재까지 업데이트합니다 [11, 122, 135]. 마지막으로, 접근 방식의 세 번째 라인은 다양한 수준의 지역으로 이벤트를 캡처하고 부분 결과를 통합하는 다중 파티션, 즉 데이터 분할을 유지합니다 [45, 99].

특히 개정 처리에는 운영자 지원, 성능 오버헤드, 개정 가능한 데이터의 최신성에 대한 제한 등 일련의 문제가 발생합니다.

개정 처리는 특히 전체 창을 다시 계산해야 하는 다운스트림 상태 저장 연산자의 경우 스트리밍 시스템에 적지 않은 처리 오버헤드를 추가할 수 있습니다. 개정 데이터가 증가하면 오버헤드가 급격히 증가할 수 있습니다. 마지막으로 원본 입력 데이터를 수정하기 위해 영원히 보관하는 것은 불가능하므로 수정 처리는 가장 최근 데이터를 포함하는 하위 집합으로 제한됩니다.

#### 3.5.3.1 저장 및 수정. Microsoft의 CEDR [31], StreamInsight [17] 및 Google의 Dataflow [16] 버퍼 또는 저장소 스트림

캡처된 값을 수정하고 계산을 업데이트하여 데이터를 수집하고 자연 이벤트, 업데이트 및 삭제를 점진적으로 처리합니다.

데이터 흐름 모델 [16]은 비순차적 데이터에 대한 우려를 늦은 데이터가 처리되는 이벤트 시간, 해당 결과가 구체화되는 처리 시간, 이후 업데이트가 이전 결과와 어떻게 관련되는지 세 가지 차원으로 나눕니다. 업데이트된 결과의 방출과 개선 방법을 결정하는 메커니즘을 트리거라고 합니다.

트리거는 지정된 규칙 집합이 실행될 때 계산이 반복되거나 업데이트되도록 하는 신호입니다.

한 가지 중요한 규칙은 늦은 입력 데이터 도착과 관련됩니다. 트리거는 늦은 입력의 효과를 계산 결과에 통합하여 출력 정확성을 보장합니다. 트리거는 워터마크, 처리 시간, 데이터 도착 지표 및 이들의 조합을 기반으로 정의할 수 있습니다. 사용자가 정의할 수도 있습니다. 트리거는 새 결과가 이전 결과를 보완하는 위치를 누적하고, 새 결과가 이전 결과를 덮어쓰는 위치를 삭제하고, 새 결과가 이전 결과를 덮어쓰고 이전 결과가 철회되는 위치를 누적하고 철회하는 세 가지 구체화 정책을 지원합니다. StreamInsight [17]에서는 철회 또는 보상도 지원됩니다.

**3.5.3.2 재생 및 수정. 동적 개정 [11] 및 추측 처리 [122]**는 개정 레코드가 수신될 때 영향을 받은 과거 데이터 하위 집합을 재생합니다. 이 체계의 최적화는 업스트림 처리와 다운스트림 처리라는 두 가지 개정 처리 메커니즘에 의존합니다 [135]. 둘 다 두 일반 연산자 사이에 개입하고 업스트림 연산자가 출력 한 레코드를 저장하는 연결 지점이라는 특수 목적 연산자를 기반으로 합니다. 업스트림 개정 처리에 따라 연결 지점의 다운스트림 운영자는 재생할 레코드 세트를 요청할 수 있으므로 이전 결과와 새 결과를 기반으로 개정을 계산 할 수 있습니다. 대안으로, 운영자는 다운스트림 연결 지점에서 수신된 개정 기록과 관련된 출력 기록 세트를 검색하도록 요청할 수 있습니다. 상황에 따라 운영자는 원본 기록과 수정된 기록 간의 차이의 순 효과를 이전 결과에 통합하여 올바른 수정본을 계산할 수 있습니다.

동적 개정은 원래 값과 개정된 값 간의 출력 차이가 포함된 델타 개정 메시지를 내보냅니다. 이 접근 방식은 입력 큐의 연결 지점에 있는 운영자의 입력 메시지 기록을 유지합니다. 모든 메시지를 보관하는 것은 불가능하기 때문에 보관되는 메시지의 기록에는 한계가 있습니다. 이 경계에서 더 뒤로 이동하는 메시지는 재생할 수 없으므로 수정될 수 없습니다. 동적 개정은 상태 비저장 연산자와 상태 저장 연산자를 구별합니다. 상태 비저장 연산자는 원본 ( $t$ ) 과 수정된 메시지 ( $t'$ )를 모두 평가하여 출력의 델타를 내보냅니다. 예를 들어, 연산자가 필터이고  $t$  가 참이고  $t'$  가 아닌 경우 연산자는  $t$ 에 대한 삭제 메시지를 내보냅니다. 반면에 상태 저장 연산자는 다음을 수행해야 합니다.

ators in the loop are able to revise the original speculative punctuation. The approach can be applied in systems that provide speculative output for strongly convergent queries, which provide finite results for finite inputs with finite derivations for each result.

In Naiad [120, 121], the general progress-tracking model features logical, i.e. processing-time, multidimensional timestamps attached to events. Each timestamp consists of the input batch to which an event belongs and an iteration counter for each loop the event traverses. As in Chandramouli et al. [46], Naiad supports cyclic queries by utilizing a special operator. However, the operator is used to increment the iteration counter of events entering a loop. To ensure progress, the system allows event handlers to dispatch only messages with larger timestamps than the timestamp of events being currently processed. This restriction imposes a partial order over all pending events. The order is used to compute the earliest logical time of events' processing completion in order to deliver notifications for producing output. Naiad's progress-tracking mechanism enables a) scalability by providing efficient delivery of notifications to dataflow nodes and b) incremental computation that avoids redundant work.

### 3.5.3 Revision processing

Revision processing is the update of computations in face of late, updated, or retracted data, which require the modification of previous outputs in order to provide correct results. Revision processing made its debut in Borealis [11]. From there on, it has been combined with in-order processing architectures [45, 122], as well as out-of-order processing architectures [16, 17, 31, 99]. In some approaches, revision processing works by *storing* incoming data and *revising* computations in face of late, updated, or retracted data [16, 17, 31]. Other approaches *replay* affected data, *revise* computations, and propagate the revision messages to update all affected results up to the present [11, 122, 135]. Finally, a third line of approaches maintain multiple *partitions*, i.e. divisions of the data, that capture events with different levels of lateness and *consolidate* partial results [45, 99].

Notably, revision processing introduces a set of challenges, namely operator support, performance overhead, and limitations regarding the recency of data that can be revised. Revision processing can add non-trivial processing overhead to a streaming system, especially for downstream stateful operators that will need to recompute entire windows. The overhead can increase sharply with the increase of revision data. Finally, since it is impossible to hold all original input data eternally to be able to revise them, revision processing is restricted to a subset containing the most recent data.

**3.5.3.1 Store and revise.** Microsoft's CEDR [31], StreamInsight [17], and Google's Dataflow [16] buffer or store stream

data and process late events, updates, and deletions incrementally by revising the captured values and updating the computations.

The dataflow model [16] divides the concerns for out-of-order data into three dimensions: the event time when late data are processed, the processing time when corresponding results are materialized, and how later updates relate to earlier results. The mechanism that decides the emission of updated results and how the refinement will happen is called a *trigger*. Triggers are signals that cause a computation to be repeated or updated when a set of specified rules fire.

One important rule regards the arrival of late input data. Triggers ensure output correctness by incorporating the effects of late input into the computation results. Triggers can be defined based on watermarks, processing time, data arrival metrics, and combinations of those; they can also be user-defined. Triggers support three refinement policies, *accumulating* where new results complement older ones, *discarding* where new results overwrite older ones, and *accumulating and retracting* where new results overwrite older ones and older results are retracted. Retractions, or compensations, are also supported in StreamInsight [17].

**3.5.3.2 Replay and revise.** Dynamic revision [11] and speculative processing [122] replay an affected past data subset when a revision record is received. An optimization of this scheme relies on two revision processing mechanisms, upstream processing and downstream processing [135]. Both are based on a special-purpose operator, called a *connection point*, that intervenes between two regular operators and stores records output by the upstream operator. According to the upstream revision processing, an operator downstream from a connection point can ask for a set of records to be replayed, so that it can calculate revisions based on old and new results. Alternatively, the operator can ask from the downstream connection point to retrieve a set of output records related to a received revision record. Under circumstances, the operator can calculate correct revisions by incorporating the net effect of the difference between the original record and its revised one to the old result.

Dynamic revision emits delta revision messages, which contain the difference of the output between the original and the revised value. This approach keeps the input message history at an operator in the connection point of its input queue. Since keeping all messages is infeasible, there is a bound in the history of messages kept. Messages that go further back from this bound cannot be replayed and, thus, revised. Dynamic revision differentiates between stateless and stateful operators. A stateless operator will evaluate both the original ( $t$ ) and the revised message ( $t'$ ), emitting the delta of their output. For instance, if the operator is a filter,  $t$  is true and  $t'$  is not, then the operator will emit a deletion message for  $t$ . A stateful operator, on the other hand, has to

출력을 방출하기 위해 많은 메시지를 처리합니다. 따라서 집계 연산자는 수정된 메시지를 내보내려면 수정된 메시지와 해당 창에 포함된 원본 메시지 모두에 대해 전체 창을 다시 처리해야 합니다. 동적 개정은 Borealis에서 구현됩니다.

반면, 추측 처리는 무질서한 입력 스트림에 대해 출력이 생성되지 않은 경우 상태 스냅샷 복구를 적용합니다. 그렇지 않으면 생성된 모든 출력이 취소됩니다. 추측 처리에서는 개정 처리가 기회주의적이므로 히스토리 경계가 설정되지 않습니다.

**3.5.3.3 분할 및 통합.** 순서 독립적 처리 [99] 및 조바심 정렬 [45]은 독립적인 데이터 파티션을 부분적으로 병렬로 처리하고 부분 결과를 통합합니다. 순서 독립적 처리 접근 방식은 해당 진행률을 표시기 이후에 레코드가 수신되면 새 파티션을 엽니다. 새로운 쿼리 계획 인스턴스는 표준 비순차적 처리 기술을 사용하여 이 파티션을 처리합니다. 조바심 정렬 접근 방식에는 각 파티션에 도착하는 입력을 점진적으로 정렬하여 순서대로 내보내는 온라인 정렬 연산자가 있습니다. 이 접근 방식은 임의로 늦게 도착하는 이벤트를 처리할 수 있는 순서 독립적 처리와 달리 구두점을 사용하여 장애를 제한합니다.

순서 독립적 처리에서는 파티셔닝이 시스템에 의해 결정되지만, 조바심 정렬에서 사용자가 지정합니다. 순서 독립적 처리에서는 너무 오래되어 원래 파티션에서 고려할 수 없는 레코드는 가장 가까운 데이터가 있는 레코드가 있는 파티션에 포함됩니다.

오랫동안 임시 파티션에 새 데이터가 입력되지 않으면 파티션이 닫히고 하트비트를 통해 파괴됩니다.

임시 파티션은 창 기반입니다. 임시 파티션 중 하나에 속하지 않는 순서가 잘못된 레코드가 수신되면 새 임시 파티션이 도입됩니다. 임시 파티션의 창보다 최신 타임 스템프가 있는 순서가 잘못된 레코드로 인해 해당 파티션이 결과를 플러시하고 닫힙니다. 순서 독립적 처리는 Truviso [99]에서 구현됩니다.

반대로 조급함 정렬에서는 사용자가 순서가 잘못된 입력 레코드를 수집하고 정렬하기 위한 버퍼링 시간을 정의하는 재정렬 대기 시간(예: 1ms, 100ms, 1s)을 지정합니다.

지정된 재정렬 대기 시간에 따라 시스템은 순서대로 입력 스트림의 다양한 파티션을 생성합니다. 정렬 후 통합 연산자는 파티션 P의 출력을 P 보다 재정렬 대기 시간이 낮은 파티션 L의 출력과 병합하고 동기화합니다. 따라서 출력에는 P에 포함된 이후 업데이트와 함께 L에서 제공한 부분 결과가 통합됩니다. 이런 방식으로 빠르지만 부분적인 결과가 필요한 애플리케이션은 재정렬 대기 시간이 짧은 파티션을 구독할 수 있습니다. 애플리케이션이 원하는 재주문 대기 시간 범위를 선택할 수 있도록 함으로써 이 설계는 결과의 완전성과 최신성 사이에서 서로 다른 절충안을 제공합니다. 조바심 정렬은 Microsoft Trill에서 구현됩니다.

### 3.6 1세대 대 2세대

데이터 스트림 처리에서 이벤트 순서의 중요성은 초기부터 명백했으며 [27], 간단하고 직관적인 솔루션의 첫 번째 물결로 이어졌습니다. 초기 접근 방식에는 스트리밍 시스템에 순서대로 전송되는 데이터의 빈도와 자연을 조정하기 위한 몇 가지 조치를 사용하여 도착 레코드를 버퍼링하고 재정렬하는 것이 포함되었습니다 [12, 48, 142]. 몇 년 후, 비순차적 처리 [108]가 도입되면서 레코드를 주문하지 않고도 처리 진행 상황을 추적함으로써 창 작업의 처리량, 대기 시간 및 확장성이 향상되었습니다. 한편, 비순차적 데이터를 반응적으로 처리하기 위한 전략으로 개정 처리 [11]가 제안되었습니다. 그 후 몇 년 동안 순차, 비순차 및 개정 처리가 광범위하게 연구되었으며 종종 서로 결합되었습니다 [16, 17, 31, 99, 122]. 최신 스트리밍 시스템은 이러한 원래 개념을 개선하여 구현합니다. 흥미롭게도 원본 개정 처리를 향상시키는 로우 워터마크, 구두점 및 트리거와 같이 몇 년 전에 고안된 개념이 최근 Millwheel [14] 및 Google Dataflow 모델 [16], Flink [37]과 같은 스트리밍 시스템에 의해 대중화되었습니다. 및 스파크 [23]. 표 1은 1세대 및 최신 스트리밍 시스템이 비순차적 데이터 관리를 구현하는 방법을 보여줍니다.

멘션.

### 3.7 미해결 문제

데이터 장애를 관리하려면 아키텍처 지원과 유연한 메커니즘이 필요합니다. 두 수준 모두에 공개된 문제가 있습니다.

첫째, 어떤 아키텍처가 더 나은지는 공개 토론입니다.

많은 최신 스트리밍 시스템이 비순차적 아키텍처를 선택하고 있지만 반대자들은 아키텍처의 구현 및 유지 관리 복잡성을 지적합니다. 또한, 순서가 잘못된 레코드를 조정하는 데 사용되는 개정 처리는 어려운 상태 크기로 인해 규모 면에서 매우 어렵습니다. 반면, 순차 처리는 리소스를 많이 소모하며 장애 경계 이후에 도착하면 이벤트를 잃습니다.

둘째, 다양한 소스에서 데이터 스트림을 수신하는 애플리케이션은 예를 들어 들어오는 스트림당 하나씩 이벤트 시간에 대한 여러 개념을 지원해야 할 수 있습니다. 그러나 현재의 스트리밍 시스템은 다중 시간 영역을 지원할 수 없습니다.

마지막으로, 다양한 소스의 데이터 스트림에는 워터마크가 정렬되지 않은 상태로 렌더링되는 서로 다른 대기 시간 특성이 있을 수 있습니다. 오늘날의 스트리밍 시스템에서는 이러한 애플리케이션의 처리 진행 상황을 추적하는 것이 어렵습니다.

## 4 상태 관리

상태는 연속 스트림 계산의 모든 내부 부작용을 캡처합니다. 상태에는 활성 창, 레코드 버킷, 부분 또는 증분 집계 등이 포함됩니다.

process many messages in order to emit an output. Thus, an aggregation operator has to re-process the whole window for both a revised message and the original message contained in that window in order to emit revision messages. Dynamic revision is implemented in Borealis.

Speculative processing, on the other hand, applies state snapshot recovery if no output has been produced for a disordered input stream. Otherwise, it retracts all produced output. In speculative processing, because revision processing is opportunistic, no history bound is set.

**3.5.3.3 Partition and consolidate.** *Order-independent processing* [99] and *impatience sort* [45] partially process independent data partitions in parallel and consolidate partial results. The order-independent processing approach opens a new partition when a record is received after its corresponding progress indicator. A new query plan instance processes this partition using standard out-of-order processing techniques. The impatience sort approach features an online sorting operator, which incrementally orders the input arriving at each partition, so that it is emitted in order. The approach uses punctuations to bound the disorder, as opposed to order-independent processing, which can handle events arriving arbitrarily late.

In order-independent processing, partitioning is left for the system to decide, while in impatience sort it is specified by the user. In order-independent processing, records that are too old to be considered in their original partition are included in the partition which has the record with the closest data. When no new data enter an ad-hoc partition for a long time, the partition is closed and destroyed by means of a heartbeat. Ad-hoc partitions are window-based; when an out-of-order record is received that does not belong to one of the ad-hoc partitions, a new ad-hoc partition is introduced. An out-of-order record with a more recent timestamp than the window of an ad-hoc partition causes that partition to flush results and close. Order-independent processing is implemented in Truviso [99].

On the contrary, in impatience sort, users specify reorder latencies, such as 1ms, 100ms, and 1s, that define the buffering time for ingesting and sorting out-of-order input records. According to the specified reorder latencies, the system creates different partitions of in-order input streams. After sorting, a union operator merges and synchronizes the output of a partition  $P$  with the output of a partition  $L$  that features lower reorder latency than  $P$ . Thus, the output will incorporate partial results provided by  $L$  with later updates that  $P$  contains. This way, applications that require fast but partial results can subscribe to a partition with small reorder latency. By letting applications choose the desired extent of reorder latency, this design provides for different trade-offs between completeness and freshness of results. Impatience sort is implemented in Microsoft Trill.

### 3.6 First generation versus second generation

The importance of event order in data stream processing was obvious since its early days [27], leading to the first wave of simple intuitive solutions. Early approaches involved buffering and reordering arriving records using some measure for adjusting the frequency and lateness of data dispatched to a streaming system in order [12, 48, 142]. A few years later, the introduction of out-of-order processing [108] improved throughput, latency, and scalability for window operations by keeping track of processing progress without ordering records. In the meantime, revision processing [11] was proposed as a strategy for dealing with out-of-order data reactively. In the years that followed, in-order, out-of-order, and revision processing were extensively explored, often in combination with one another [16, 17, 31, 99, 122]. Modern streaming systems implement a refinement of these original concepts. Interestingly, concepts devised several years ago, like low-watermarks, punctuations, and triggers, which advance the original revision processing, were popularized recently by streaming systems such as Millwheel [14] and the Google Dataflow model [16], Flink [37], and Spark [23]. Table 1 presents how both first generation and modern streaming systems implement out-of-order data management.

### 3.7 Open problems

Managing data disorder entails architecture support and flexible mechanisms. There are open problems at both levels.

First, which architecture is better is an open debate. Although many of the latest streaming systems adopt an out-of-order architecture, opponents point to the architecture's implementation and maintenance complexity. In addition, revision processing, which is used to reconcile out-of-order records is daunting at scale because of the challenging state size. On the other hand, in-order processing is resource-hungry and loses events if they arrive after the disorder bound.

Second, applications receiving data streams from different sources may need to support multiple notions of event time, one per incoming stream, for instance. However, streaming systems to date cannot support multiple time domains.

Finally, data streams from different sources may have disparate latency characteristics that render their watermarks unaligned. Tracking the processing progress of those applications is challenging for today's streaming systems.

## 4 State management

State captures all internal side-effects of a continuous stream computation. The state includes, for example, active windows, buckets of records, partial or incremental aggregates

표 1 스트리밍 시스템의 이벤트 순서 관리

체계	건축학		진행 상황 추적			개정 접근하다
	순서대로	고장난	개정	기구	의사소통 장애	
오로라 [12, 52]			느슨하게	사용자 구성	숫자 기록의	-
스트림 [142]			하트비트	신호를 보내다 입력엔- 노화	타임스탬프 (이벤트 시간 왜곡, 그물- 일하다 지연 시간, 밖의 주문하다 경계)	-
보레알리스 [11]			역사 바운드	시스템 구성	숫자 기록의 또는 시간 단위	과거 데이터를 재생하고, 수정된 값 입력, 델타 발행
기가스코프 [87]	로우 워터마크	구두			타임스탬프	-
타임스트림 [129]	로우 워터마크	구두			타임스탬프	-
밀월 [14]	로우 워터마크	중앙 당국에 신호			타임스탬프	-
나이아드 [120]	포인트 스템프	데이터의 일부 기록		다차원 시셔널 타임스탬프	증분 처리 업데이트된 데이터 구조화를 통해 루프	
트릴 [44]	로우 워터마크	구두			타임스탬프	-
스트림스코프 [109]	로우 워터마크	구두			타임스탬프; 순서 숫자	-
삼자 [124]	로우 워터마크	구두			타임스탬프	찾기, 굽리기 뒤로, 다시 계산 에 영향을 받음 입력 창
좋아요 [37]	로우 워터마크	구두			타임스탬프	가게 추천- 냄새/수정
데이터플로우 [16]	로우 워터마크	중앙 당국에 신호			타임스탬프	폐기하고 재계산됨; 축적하다 수정하고; 관습
스파크 [23]	느슨하게	사용자 구성	숫자 초		폐기하고 재계산됨; 축적하다 그리고 수정하다	

**Table 1** Event order management in streaming systems

System	Architecture		Progress-tracking				
	In-order	Out-of-order	Revision	Mechanism	Communication	Disorder bound metric	Revision approach
Aurora [12, 52]	✓			Slack	User config	Number of records	–
STREAMS [142]	✓			Heartbeat	Signal to input manager	Timestamp (event time skew, net-work latency, out-of-order bound)	–
Borealis [11]		✓	✓	History bound	System config	Number of records or time units	Replay past data, enter revised values, issue delta output
Gigascope [87]		✓		Low-watermark	Punctuation	Timestamp	–
Timestream [129]	✓			Low-watermark	Punctuation	Timestamp	–
Millwheel [14]		✓		Low-watermark	Signal to central authority	Timestamp	–
Naiad [120]		✓	✓	Pointstamp	Part of data record	Multidimensional timestamp	Incremental processing of updated data via structured loops
Trill [44]	✓			Low-watermark	Punctuation	Timestamp	–
Streamscope [109]	✓			Low-watermark	Punctuation	Timestamp; sequence number	–
Samza [124]		✓	✓	Low-watermark	Punctuation	Timestamp	Find, roll back, recompute affected input windows
Flink [37]	✓	✓	✓	Low-watermark	Punctuation	Timestamp	Store and Recompute/Revise
Dataflow [16]	✓	✓	✓	Low-watermark	Signal to central authority	Timestamp	Discard and recompute; accumulate and revise; custom
Spark [23]	✓	✓	✓	Slack	User config	Number of seconds	Discard and recompute; accumulate and revise

애플리케이션에서 사용되며 스트림 파이프라인 실행 중에 생성 및 업데이트되는 일부 사용자 정의 변수도 포함됩니다. 이 섹션에서는 상태 관리의 맥락에서 알려진 접근 방식, 현재 방향 및 미해결 문제에 대한 논의에 대한 개요를 제공합니다.

## 4.1 스트림 처리 상태 관리

스트림 상태 관리 분야는 스트림 애플리케이션에서 상태를 선언하는 방법과 상태를 확장하고 분할하는 방법에 대한 방법을 통합하는 여전히 활발한 연구 분야입니다. 또한 상태 관리는 장기 실행 애플리케이션에 대한 상태 지속성 방법을 고려하고 시스템 보증 및 속성을 정의하여 시스템에 장애나 재구성(예: 특정 연산자의 병렬 처리 수준 변경)과 같은 변경이 발생할 때마다 유지하도록 합니다.

상태 분할 상태는 여러 키에 걸쳐 계산을 병렬화하기 위해 분할되어야 합니다. 예를 들어, 한 달 동안 특정 고객의 주문 수를 생각해 보세요. 이 경우 고객의 상태는 주문 수입니다. 병렬 스트리밍 애플리케이션은 다양한 고객을 여러 작업자로 분할합니다.

재구성 중 상태 변경 스트리밍 애플리케이션은 본질적으로 장기간에 걸쳐 지속적으로 실행되어야 합니다. 그러나 장기 실행 중에는 많은 문제가 발생할 수 있습니다. 첫째, 입력 데이터의 통계(예: 키 베포) 또는 들어오는 스트림의 입력 처리량이 변경될 수 있습니다. 반대의 문제, 즉 필요한 것보다 더 많은 자원을 할당하는 것도 과도한 자원 활용으로 이어지기 때문에 문제가 됩니다. 마지막으로 오류가 발생할 수 있습니다. 며칠 또는 몇 달 동안 지속적으로 데이터를 처리하는 응용 프로그램의 경우 서버 오류(예: 디스크 또는 네트워크 오류)가 발생할 가능성이 매우 높습니다. 이러한 오류에는 상태 재구성이 필요합니다. 일반적으로 특정 키를 다른 노드에 할당하여 작업 부하의 균형을 맞추고 리소스의 과다 또는 과소 활용을 방지해야 합니다. 우리는 섹션에서 상태 재구성에 대한 접근 방식을 검토합니다. [6.3.2](#) 상태 관리 결정이 섹션의 재구성 메커니즘 설계에 어떻게 영향을 미칠 수 있는지 논의합니다. [7.1](#).

이러한 연구 문제의 대부분은 부분적으로 Aurora 및 Borealis와 같은 초기 스트리밍 처리 시스템의 맥락에서 소개되었습니다 [\[41\]](#). 특히 Borealis는 암베디드 상태, 영구 저장소 액세스 및 오류 복구 프로토콜의 필요성과 같은 많은 문제를 공식화하는 기반을 설정했습니다. 표 2에서는 데이터 스트림 처리 시스템을 각각의 상태 관리 접근 방식에 따라 분류합니다. 이 섹션의 나머지 부분에서는 과거 및 현재 사용되는 접근 방식과 함께 스트림 상태 관리의 각 주제에 대한 개요를 제공합니다.

## 4.2 프로그래밍 가능성과 책임

프로그래밍 모델의 상태는 암시적으로 또는 명시적으로 선언되고 사용될 수 있습니다. 우리는 상태 프로그래밍 가능성을 사용자가 상태를 선언하고 조작할 수 있도록 하는 스트리밍 시스템의 능력으로 정의합니다. 예를 들어, 상태는 카운터를 저장하는 상태 저장 맵 함수 내의 지역 변수일 수 있습니다. 상태의 프로그래밍에는 엔진의 복잡성에 직접적인 영향을 미치는 기능인 기본 실행 엔진의 지원이 필요합니다. 다양한 시스템 추세는 데이터 스트림 프로그래밍 모델에서 상태를 노출하는 방법과 범위를 지정하고 관리하는 방법 모두에 영향을 미쳤습니다. 이 섹션에서는 다양한 접근 방식과 그 장단점에 대해 논의합니다. 표 2에 표시된 것처럼 대부분의 시스템에서는 사용자가 사용자 정의 상태를 선언할 수 있습니다. 그렇지 않은 경우에는 내부 운영자만이 상태 저장 작업(예: 조인, 창, 집계) 내에서 상태를 정의하고 사용할 수 있도록 허용하는 데이터 흐름 프로세서 위에 상위 수준 SQL 인터페이스를 제공하는 데 더 중점을 둡니다.

상태 관리 책임 프로그래밍 가능성과 직교하는 측면은 사용자나 시스템이 상태를 유지해야 하는 의무를 수반하는 상태 책임입니다. 상태 유지 관리에는 애플리케이션 변수를 저장하기 위한 메모리 또는 디스크 공간 할당, 디스크 변경 사항 유지, 시스템 복구 시 필요한 경우 내구성 있는 저장소에서 상태 항목 복구가 포함됩니다. Storm [\[152\]](#) 및 S4 [\[123\]](#) 와 같은 1세대 데이터 병렬 스트림 처리 시스템에는 사용자 관리 상태가 필요했습니다. 이러한 시스템에서 상태 저장 처리는 보장 없이 구현되어 사용자 지정 인메모리 데이터 구조를 사용하거나 특정 확장성 및 지속성 요구 사항을 충족하는 외부 키-값 저장소를 사용하여 구현되는 경우가 많습니다. 사용 가능한 나머지 시스템의 경우 상태 관리 문제는 명시적 상태 API 또는 프로그래밍 할 수 없지만 내부적으로 관리되는 상태 추상화를 사용하여 스트리밍 시스템 자체에서 내부적으로 처리되었습니다.

### 4.2.1 토론

주 메모리가 부족했던 초기 데이터 스트림 관리에서 상태는 STREAM [\[19\]](#)에 사용된 CQL의 조인, 필터 및 정렬과 같은 시스템 연산자의 구현을 지원하는 측면 역할을 했습니다. 우리는 주어진 시스템의 설계자가 정의하고 해당 시스템의 내부 운영자가 사용하는 이러한 유형의 상태를 시스템 정의 상태라고 부릅니다. 이러한 유형의 상태를 설명하는 데 사용되는 일반적인 용어는 "시냅시스"였습니다. 일반적으로 이러한 시스템의 사용자는 기본 상태를 인식하지 못했으며 그 암시적 특성은 DBMS의 중간 결과를 사용하는 것과 유사했습니다. STREAM 및 Aurora Borealis [\[41\]](#) 와 같은 시스템은 창 최대값과 같은 다양한 연산자를 지원하는 스트림 애플리케이션의 데이터 흐름 그래프에 특수 시냅시스를 첨부했습니다.

used in an application, as well as possibly some user-defined variables created and updated during the execution of a stream pipeline. This section provides an overview of known approaches, current directions, and discussions of open problems in the context of state management.

## 4.1 Managing stream processing state

The area of stream state management is still an active research field, incorporating methods on how state should be declared in a stream application, as well as how it should be scaled and partitioned. Furthermore, state management considers state persistence methods for long-running applications, and defines system guarantees and properties to maintain whenever a change in the system occurs, such as failures or reconfiguration, e.g., changing the degree of parallelism of a given operator.

**State partitioning** State needs to be partitioned in order to parallelize computations across different keys. For instance, consider the count of orders of a given customer, during a time window of a month. The state of the customer in this case is the count of orders. A parallel streaming application would partition the different customers on multiple workers.

**State changes during reconfiguration** Streaming applications have an inherent need to run continuously over long periods of time. However, during long executions, a lot of issues may arise. First, the statistics of the input data (e.g., distribution of keys) or the input throughput of the incoming stream may change. The opposite problem, i.e., allocating more resources than needed, is also problematic because it leads to excessive resource utilization. Finally, failures can happen: the probability of servers failing (e.g., disk or network failures) is very high for an application that processes data continuously for days or months. These failures, require state reconfiguration: typically specific keys, need to be assigned to different nodes to balance the workload, avoiding over- or under-utilization of resources. We review approaches to state reconfiguration in Sect. 6.3.2 and discuss how state management decisions may affect the design of reconfiguration mechanisms in Sect. 7.1.

Most of these research issues were introduced in part within the context of early streaming processing systems, such as Aurora and Borealis [41]. Specifically, Borealis set the foundations in formulating many of these problems, such as the need for embedded state, persistent store access, and failure recovery protocols. In Table 2, we categorize data stream processing systems according to their respective state management approaches. The rest of this section offers an overview of each of the topics in stream state management along with past and currently employed approaches.

## 4.2 Programmability and responsibility

State in a programming model can be either implicitly or explicitly declared and used. We define *state programmability* as the ability of a streaming system to allow its users to declare and manipulate state. For example, state can be a local variable within a stateful map function, storing a counter. Programmability in state requires support from the underlying execution engine, a feature that directly affects the engine’s complexity. Different system trends have influenced both how state can be exposed in a data stream programming model, as well as how it should be scoped and managed. In this section, we discuss different approaches and their trade-offs. As shown in Table 2, most systems allow their users to declare custom, user-defined state. Those that do not, focus more on providing a high-level SQL interface on top of a dataflow processor allowing only their internal operators to define and use state within stateful operations (e.g., joins, windows, aggregates).

**State management responsibility** An orthogonal aspect to programmability is state management *responsibility*, which entails the obligation of maintaining state by either the user or the system. State maintenance includes allocating memory or disk space for storing application variables, persisting changes to disk and recovering state entries from durable storage if needed upon system recovery. The first generation of data-parallel stream processing systems, such as Storm [152] and S4 [123], required user-managed state. In such systems, stateful processing was either implemented with no guarantees, making use of custom in-memory data structures or, often implemented using external key-value stores that cover certain scalability and persistence needs. For the rest of the systems available, state management concerns have been internally handled by the streaming systems themselves through the use of explicit state APIs or non-programmable, yet internally managed, state abstractions.

### 4.2.1 Discussion

In the early days of data stream management when main memory was scarce, state had a facilitating role, supporting the implementation of system operators, such as CQL’s join, filter, and sort as employed in STREAM [19]. We term this type of state, defined by the designers of a given system and used by the internal operators of that system, *system-defined state*. A common term used to describe that type of state was “synopsis”. Typically, users of such systems were oblivious of the underlying state and its implicit nature resembled the use of intermediate results in DBMSs. Systems such as STREAM, as well as Aurora Borealis [41], attached special synopses to a stream application’s dataflow graph supporting different operators, such as a window max,

스트림의 진화에 관한 조사…

표 2 스트리밍 시스템의 상태 관리 기능

체계	프로그래밍 가능한 상태 상태 관리 책임 상태 관리 아키텍처	저장매체
오로라/보레알리스 [52]	체계	인 메모리 아웃 오브 코어 외부 탄력성 저장소 임시 없음 로컬 원격
스트림 [19]	체계	
TelegraphCQ [136]	체계	
S4 [123]	사용자	
스톰(1.0) [152]	사용자	
스파크(1.0) [23]	체계	
트라이던트 [9]	체계	
스킵 [65]	체계	
나이아드 [120]	체계	
타임스트림 [129]	체계	
밀월 [14]	체계	
플링크 [36, 37]	체계	
카프카 스트림 [5]	체계	
삼자 [124]	체계	
스트림스코프 [109]	체계	
S스토어 [42, 147]	체계	

오프셋을 위한 조인 인덱스 또는 입력 소스 버퍼. 주목할만한 STREAM의 기능은 시놉시스를 재사용하는 기능이었습니다. 애플리케이션에서 다른 개요를 구성적으로 정의하기 위해 시스템 내부적으로. 전반적으로 시놉시스는 다음 중 하나였습니다.

초기 스트림 처리 시스템의 첫 번째 형태의 상태, 주로 공유 메모리를 통한 스트림 처리에 사용됩니다. 내결함성을 포함한 상태와 관련된 여러 가지 문제

로드 밸런싱은 그 당시 이미 고려된 사항이었습니다. 보레알리스의 예. 그러나 사용자 정의 상태가 부족합니다. 해당 시스템 세대의 표현력이 제한됨

관계 연산의 하위 집합입니다. 또한, 지나치게 전문화된 데이터 구조는 상태가 유연하고 분할하기 쉬워야 하는 재구성 요구 사항.

MapReduce 이후 시대에는 주요 초점이 있었습니다. Storm [3] 과 같은 시스템을 사용하면 작업의 분산 파이프라인 구성을 허용하는 컴퓨팅 확장성이 있습니다. . 을 위한 애플리케이션 유연성과 단순성, 이러한 시스템 중 다수 상태 관리를 제공하지 않았으며 두 선언을 모두 남겼습니다. 프로그래머에 대한 상태 관리. 사용자 선언 및 관리 상태는 호스팅 프레임워크에서 제공하는 작업 메모리 및 범위 내에서 정의되고 사용되었습니다. 기존 키-값을 사용하여 외부에서 정의되고 지속됩니다. 스토리지 또는 데이터베이스 시스템(예: Redis [7, 106]). 요약하자면, 애플리케이션 관리 상태는 유연성을 제공하고 전문가에게 사용자 구현의 자유. 그러나 시스템 측에서는 상태 관리 지원이 제공되지 않습니다. 결과적으로, 사용자는 지속성에 대해 추론해야 합니다. 메인 메모리 및 필요한 모든 타사 스토리지에 적합

시스템 종속성. 이러한 선택에는 조합이 필요합니다. 스트림 및 스토리지 기술을 통합하기 위한 깊은 전문 지식과 추가 엔지니어링 작업이 필요합니다.

현재 대부분의 스트림 처리 시스템은 레벨을 허용합니다. 상태 저장 형식을 통해 사용자 정의 상태에 대한 자유도 확보 처리 API. 이 기능을 사용하면 스트리밍 애플리케이션이 가능해집니다. 사용자 정의 상태를 정의하는 동시에 기본 시스템에 상태 정보에 대한 액세스 권한을 부여하여 지속성과 확장성을 위한 데이터 관리 메커니즘 그리고 내결함성. 상태 정보에는 데이터 유형이 포함됩니다. 사용, 직렬 변환기 및 역직렬 변환기는 물론 읽기 및 쓰기도 가능합니다. 런타임에 알려진 작업. 사용자 정의, 시스템 관리 상태의 주요 제한 사항은 직접적인 제어가 부족하다는 것입니다. 해당 상태를 구체화하는 데이터 구조(예: 사용자 정의의 경우) 최적화).

#### 4.3 상태 관리 아키텍처

상태 관리 아키텍처는 다음과 같은 방식을 나타냅니다. 스트리밍 시스템은 내부 또는 사용자 정의 상태를 저장하고 관리합니다. 우리는 세 가지 별개의 상태 저장 처리를 식별합니다. 데이터 스트림 런타임 시스템 아키텍처의 방향:

- 인메모리 아키텍처는 인메모리를 사용하여 상태를 저장합니다. 데이터 구조. 이 접근 방식은 다음과 같은 상태를 지원할 수 있습니다. 실행 중인 각 노드에서 사용 가능한 주 메모리 내에 있습니다. 스트림 연산자.
- 외부 메모리 아키텍처는 비휘발성 메모리 또는

**Table 2** State management features in streaming systems

System	Programmable state	State Mgmt responsibility	State Mgmt architecture			Storage medium		
			In-memory	Out-of-core	External	Resilient store	Ephemeral	None
			Local	Remote				
Aurora/Borealis [52]	X	System	✓				✓	
STREAM [19]	X	System	✓			✓		✓
TelegraphCQ [136]	X	System	✓			✓		
S4 [123]	✓	User			✓			✓
Storm (1.0) [152]	✓	User			✓			✓
Spark(1.0) [23]	✓	System	✓			✓		
Trident [9]	✓	System	✓	✓			✓	
SEEP [65]	✓	System	✓			✓		
Naiad [120]	✓	System	✓			✓		
TimeStream [129]	✓	System	✓			✓		
Millwheel [14]	✓	System			✓		✓	
Flink [36, 37]	✓	System	✓	✓		✓		
Kafka-Streams [5]	X	System	✓	✓			✓	
Samza [124]	✓	System	✓	✓		✓		
Streamscope [109]	✓	System	✓			✓		
S-Store [42, 147]	X	System			✓	✓		✓

a join index or input source buffers for offsets. A noteworthy feature in STREAM was the capability to re-use synopses compositionally to define other synopses in an application internally in the system. Overall, synopses have been one of the first forms of state in early stream processing systems, primarily for stream processing over shared-memory. Several of the issues regarding state, including fault tolerance and load balancing, were already considered back then, for example in Borealis. However, the lack of user-defined state limited the expressive power of that generation of systems to a subset of relational operations. Furthermore, the use of over-specialized data structures was somewhat oblivious to the needs of reconfiguration, which requires state to be flexible and easy to partition.

In the post-MapReduce era, there was a primary focus in compute scalability with systems like Storm [3] allowing the composition of distributed pipelines of tasks. For application flexibility and simplicity, many of these systems did not offer state management, leaving both declaration and management of state to programmers. User-declared and managed state was either defined and used within the working memory and scope provided by the hosting framework or defined and persisted externally, using an existing key-value storage or database system (e.g. Redis [7, 106]). In summary, application-managed state offers flexibility and gives expert users implementation freedom. However, no state management support is offered from the system's side. As a result, the user has to reason about persistence, whether the state fits in the main-memory, and all necessary third-party storage

system dependencies. These choices require a combination of deep expertise, and additional engineering work to integrate stream and storage technologies.

Currently, most stream-processing systems allow a level of freedom for user-defined state through a form of a stateful processing API. This feature enables streaming applications to define custom state, while also granting the underlying system access to state information in order to employ data management mechanisms for persistence, scalability and fault tolerance. State information includes the data types used, serializers and deserializers as well as read and write operations known at runtime. The main limitation of user-defined, system-managed state is the lack of direct control on data structures that materialize that state (e.g., for custom optimizations).

### 4.3 State management architecture

The state management architecture refers to the way that a streaming system stores and manages its internal or user-defined state. We identify three distinct stateful processing directions in the architecture of data stream runtime systems:

- *In-memory* architectures store state using in-memory data structures. This approach is able to support state that is within main-memory available in each node executing stream operators.
- *External memory* architectures make use of multiple levels of storage media, such as non-volatile memory or

상태와 프로세스를 저장하기 위한 하드 디스크, 즉 스트림 연산자의 주소 공간 외부에 있는 메모리를 사용합니다. "아웃 오브 코어(out-of-core)"라는 용어는 외부 메모리를 기반으로 구축된 데이터 구조, 알고리즘 및 내장 데이터베이스를 설명하는 데에도 자주 사용됩니다. 이 접근 방식을 사용하면 각 컴퓨팅 노드 내에서 빠른 기본 메모리 액세스를 활용하는 동시에 보조 스토리지에 분할되어 보관되는 점점 더 많은 상태 항목을 지원할 수 있습니다. 우리는 선택한 코어 외부 데이터 구조가 사용되는 것을 관찰합니다.

대부분의 시스템에서는 FASTER [47] 또는 RocksDB/LevelDB와 같은 LSM-Tree [125] 와 같은 인덱스의 변형이 있습니다. 1 - 원격 메모리 아키텍처는 컴퓨팅과 상태를 분리하여 상태 처리를 외부 데이터베이스 또는 키로 오프로드합니다. -가치 저장소. 이 접근 방식을 사용하면 보다 모듈식 시스템 설계(마우 클라우드 친화적인 상태 및 컴퓨팅 분리)와 데이터베이스 시스템의 여러 원하는 속성(예: ACID 트랜잭션, 일관성 보장, 자동 크기 조정)을 효과적으로 재사용할 수 있습니다. 데이터 스트리밍 측면에서 더 복잡한 보장이 필요합니다. 외부 상태의 사용은 Apache Storm의 애플리케이션 내에서 주로 이루어졌습니다. 시스템 관리 상태가 없기 때문에 사용자는 모든 상태를 외부 시스템에 저장해야 합니다. 이 아키텍처에서는 상태 액세스가 필요할 때 스트리밍 운영자가 외부 시스템에 연결해야 하므로 대기 시간이 늘어납니다.

Beam/Google Dataflow의 클라우드 엔진인 Google의 Millwheel은 시스템 관리 외부 상태 아키텍처의 대표적인 예입니다. Millwheel은 BigTable [50] 및 Spanner [53] 의 기능을 기반으로 구축되었습니다 (예: 블라인드 원자 쓰기). Millwheel의 작업은 사실상 상태가 없습니다. 최근 로컬 변경 사항을 메모리에 유지하지만 전반적으로 모든 단일 출력 및 상태 업데이트를 BigTable에 단일 트랜잭션으로 커밋합니다. 이 동작은 Millwheel이 키당 모든 단일 작업 상태를 지속하기 위해 외부 저장소를 사용하고 복구 및 비며등 업데이트에 필요한 모든 필수 로그 및 검사점을 사용하고 있음을 의미합니다.

#### 4.3.1 저장매체

기본 내결합성 메커니즘을 보조하는 상태 저장 스트리밍의 또 다른 측면은 복구 및 재구성에 사용되는 상태 관리입니다. 섹션에 표시된대로, 5.1.1 다양한 옵션이 있습니다. 복구 상태는 각 상태 저장 운영자의 로컬 또는 원격 복원 저장소에 있는 복원 저장소를 사용하는 것이 좋습니다. Aurora 및 Borealis [12, 41] 의 경우 복구 상태는 탄력성이 없는 임시 공간(예: 운영자 프로세스 메모리)에서 유지됩니다. 출력 풀과 같이 메모리 복구를 위해 데이터를 캐시하는 시스템은 이 범주에 속하지 않습니다.

<sup>1</sup> <https://www.github.com/google/leveldb>.

#### 4.3.2 토론

스트림 처리는 확장 가능한 컴퓨팅의 일반적인 추세에 영향을 받았습니다. 상태 및 컴퓨팅은 확장 작업 병렬 실행 모델에서 사용할 수 있는 상태 표현 및 작업에 관련 영향을 미치는 보다 일반적인 확장 데이터 병렬 모델로 점진적으로 발전했습니다. 영구 데이터 구조는 데이터베이스 관리 시스템이 고안된 이후부터 널리 사용되어 왔습니다. 내부 및 외부 지속성 전략을 사용한다는 아이디어는 최신 세대의 시스템에서 균일하게 수용되었습니다. 섹션 4.4 에서는 다양한 아키텍처를 다루고 현대 시스템이 무제한 실행 내에서 메모리에 들어갈 수 있는 수준을 넘어서는 대용량 상태를 어떻게 지원할 수 있는지에 대한 예를 제시합니다. 스트림 기술의 또 다른 기본 전환 단계는 거래 수준 보장의 개발 및 채택이었습니다. 섹션 5.1.1 에서는 최신 기술에 대한 개요를 제공하고 구현 방법론과 함께 데이터 스트리밍 트랜잭션의 의미를 다룹니다.

### 4.4 확장성과 상태 관리

확장 가능한 상태는 두 번째의 주요 인센티브였습니다.

데이터 스트림 계산의 배포 및 분할을 자동화하는 스트림 처리 시스템 생성.

확장 가능한 상태에 대한 필요성은 제한되지 않은 방대한 데이터 스트림의 가능성으로 인해 발생했습니다. 대용량 스트리밍 계산에서 스트림 상태의 공간 복잡성은 스트림 프로세서에서 소비하는 계속 증가하는 입력에 선형입니다. 이 섹션에서는 확장 가능한 상태 유형뿐만 아니라 대규모 상태에 대한 변경 사항 분할, 지속 및 커밋에 대한 지원을 유지할 수 있는 확장 가능한 시스템 아키텍처에 대해 설명합니다.

#### 4.4.1 병렬 및 전역 상태 저장 작업

상태 저장 계산(예: 주어진 키에 대한 집계)에서 데이터 병렬성을 사용하려면 계산 상태도 여러 연산자 인스턴스에 걸쳐 분할되어야 합니다.

그러나 분할 상태가 항상 가능한 것은 아닙니다(예: 스트림의 모든 키에 대해 집계를 수행해야 하는 경우).

확장 가능 상태는 스트리밍 애플리케이션에서 일반적으로 분할된 상태 와 분할되지 않은 상태( 전역 상태 라고도 함 )라는 두 가지 형태를 취합니다. 특정 작업의 성격에 따라 이러한 상태 유형 중 하나 또는 둘 다를 사용할 수 있습니다.

분할 상태 분할 상태는 대규모 데이터 스트림에서 데이터 병렬 계산을 활성화하는 가장 일반적인 방법입니다.

분할된 상태는 상태의 키별 논리적 파티션을 컴퓨팅 작업에 할당하며, 여기서 각 논리적 작업은 특정 키를 처리합니다. 이 접근 방식은 API 수준에서 다음을 통해 활성화됩니다.

hard disks to store state and process, i.e., using memory outside the address space of a stream operator. The term “out-of-core” is also frequently used to describe data structures, algorithms, and embedded databases that build on external memory. This approach allows exploiting fast main-memory access within each compute node, while also supporting a growing number of state entries that are split and archived in secondary storage. We observe that the out-of-core data structure of choice used in most systems is a variant of an index, such as FASTER [47] or an LSM-Tree [125], such as RocksDB/LevelDB.<sup>1</sup>

- *Remote memory* architectures decouple compute and state, offloading state handling to an external database or key-value store. This approach enables more modular system designs (state and compute decoupling which is very Cloud-friendly) and effective re-use of several desired properties of database systems (e.g., ACID transactions, consistency guarantees, auto-scaling) in support of more complex guarantees in the context of data streaming. The use of external state was predominant within applications in Apache Storm. The lack of system-managed state necessitated users to store all of their state in an external system. In this architecture, when state access is needed, the streaming operator has to reach out to the external system, increasing its latency. Google’s Millwheel, the cloud engine of Beam/Google Dataflow, is a representative example of system-managed external state architecture. Millwheel builds on the capabilities of BigTable [50] and Spanner [53] (e.g., blind atomic writes). Tasks in Millwheel are effectively stateless. They do keep recent local changes in memory but overall they commit every single output and state update to BigTable as a single transaction. This behavior means that Millwheel is using an external store for both persisting every single working state per key but also all necessary logs and checkpoints needed for recovery and non-idempotent updates.

#### 4.3.1 Storage medium

Another aspect of stateful streaming, auxiliary to the underlying fault-tolerance mechanisms, is the management of state used for recovery and reconfiguration. As shown in Sect. 5.1.1 there are different options. Recovery state is preferably making use of a *resilient store* that is either *local* to each stateful operator, or in a *remote* resilient store. In the case of Aurora and Borealis [12, 41], recovery state is maintained in non-resilient ephemeral space (e.g., operator process memory). Systems that cache data for recovery in memory, such as output tuples, do not fall in this category.

<sup>1</sup> <https://www.github.com/google/leveldb>.

#### 4.3.2 Discussion

Stream processing has been influenced by general trends in scalable computing. State and compute have gradually evolved from a scale-up task-parallel execution model to the more common scale-out data-parallel model with related implications in state representations and operations that can be employed. Persistent data structures have been widely used in database management systems ever since they were conceived. The idea of employing internal and external persistence strategies was uniformly embraced in more recent generations of systems. Section 4.4 covers different architectures and presents examples of how modern systems can support large volumes of state, beyond what can fit in memory, within unbounded executions. Another foundational transitioning step in stream technology was the development and adoption of transactional-level guarantees. Section 5.1.1 gives an overview of the state of the art and covers the semantics of transactions in data streaming, alongside implementation methodologies.

### 4.4 Scalability and state management

Scalable state has been the main incentive of the second generation of stream processing systems which automated deployment and partitioning of data stream computations. The need for scalable state was driven by the availability of voluminous unbounded data streams. In high-volume streaming computations, the space complexity for stream state is linear to the ever-increasing input consumed by a stream processor. This section discusses types of scalable state, as well as scalable system architectures that can sustain support for partitioning, persisting, and committing changes to large volumes of state.

#### 4.4.1 Parallel versus global stateful operations

To employ data-parallelism in a stateful computation (e.g., an aggregate on a given key) the state of the computation also has to be partitioned across different operator instances. However, partitioning state is not always possible (e.g., when an aggregate has to be performed across all keys of a stream). Scalable state takes two forms in a streaming application, typically referred to as *partitioned* and *non-partitioned* state (also referred to as *global* state). Depending on the nature of a specific operation, one or both of these state types can be employed.

**Partitioned state** Partitioned state is the most common way to enable data-parallel computation on massive data streams. Partitioned state assigns key-wise logical partitions of state to compute tasks, where each logical task handles a specific key. This approach is enabled in the API level through an

Apache Flink의 "keyBy" 또는 Beam 및 Kafka-Streams의 "groupBy"와 같이 작업 기반 처리에서 키 기반 처리로 범위를 높이는 상태 저장 처리 이전에 호출되는 추가 작업입니다. 논리적 분할(즉, 어떤 논리 연산자가 주어진 키를 인수하는지)은 물리적 분할(즉, 어떤 노드가 주어진 키 집합에 대한 계산을 인수하는지)과 다릅니다. 일반적으로 특정 컴퓨팅 노드에 여러 키(또는 키 범위)가 할당됩니다.

분할되지 않은 상태 분할되지 않은 상태는 물리적 컴퓨팅 작업에 싱글톤으로 매픾됩니다. 이러한 분할되지 않은 상태는 일반적으로 두 가지 방식으로 사용됩니다. 첫째, 전체 입력 스트림에 대한 전역 집계를 계산하는 데 사용할 수 있습니다. 둘째, 물리 연산자 수준에서 집계를 계산하는 데 사용할 수 있습니다(예: 연산자당 처리된 키 수 계산). 작업 수준 상태는 물리적 스트림 소스 작업에서 그를 사용할 때 오프셋을 유지하는 데에도 유용할 수 있습니다. 분할되지 않은 상태는 운영자-로컬 계산 또는 전역 집계를 처리하므로 해당 사용은 확장 가능하지 않으며 실무자는 주의해서 사용해야 합니다.

## 4.5 1세대 대 2세대

상태는 스트림 처리의 중심이었습니다. 상태 자체의 개념은 "요약", "시놉시스", "스케치" 또는 "스트림 테이블"과 같은 많은 이름으로 다루어져 왔으며 수년에 따른 데이터 스트림 관리의 발전을 반영합니다. 초기 시스템 [12, 19, 27, 48] (2000~2010년경)은 사용자로부터 상태와 관리를 숨겼습니다. STREAM [19] 의 CQL [21] 시변 관계형 모델과 같은 당시 대부분의 연속 처리 연산자는 내부 메모리 내 데이터 구조를 사용하여 구현되었습니다.

전반적으로 상태의 목적은 각 시스템에서 제공하는 제한된 운영자 집합의 생성을 지원하는 것이었습니다.

10년 후 MapReduce [61] 아키텍처를 기반으로 하는 확장 가능한 데이터 컴퓨팅 시스템은 분산 미들웨어 및 분할된 파일 시스템을 사용하여 임의의 사용자 정의 논리를 안정적으로 확장하고 실행할 수 있게 되었습니다. 동일한 추세에 따라 많은 기존 데이터 관리 모델이 확장성을 염두에 두고 재검토되고 재구축되었습니다(예: NoSQL 및 NewSQL 데이터베이스). 마찬가지로, 점점 더 많은 확장 가능한 데이터 스트림 처리 시스템 [14, 16, 37, 118]은 과거에 식별된 스트림 의미론 및 모델과 확장 가능한 컴퓨팅 원칙을 결합했습니다(예: 비순차적 처리 [108, 142]).

오늘날 최신 스트림 프로세서는 명확한 트랜잭션 보장 세트를 통해 내결합성과 재구성이 가능한 완전한 사용자 정의(시스템 관리) 상태로 장기 실행 연산자의 그래프를 컴파일하고 실행할 수 있습니다[14, 36, 64].

## 4.6 미해결 문제

데이터 스트리밍은 스트림 처리 기술의 원래 목적이었던 실시간 분석을 넘어서는 오늘날의 많은 데이터 관리 요구 사항을 다루고 있습니다. 상태 저장 처리의 전환은 이러한 추세를 보여줍니다.

상태 지속성과 상태 프로그래밍의 분리는 데이터베이스의 데이터 독립성 개념과 유사합니다. 시스템은 상태에 대한 의미 및 작업 측면에서 수렴되고 있으며 동시에 임베디드 데이터베이스에 사용되는 많은 새로운 방법(예: LSM 트리, 상태 인덱싱, 외부화된 상태)은 스트림 프로세서가 성능 측면에서 발전하는 데 도움이 됩니다. 능력. 최근 연구 [24, 88, 105]에서는 워크로드 인식 상태 관리, 상태 지속성 조정 및 데이터 흐름 그래프의 개별 연산자에 대한 액세스의 잠재력을 보여줍니다. 이를 위해 다양한 기능을 갖춘 로컬 상태 관리를 위한 점점 더 많은 "플러그형" 시스템 [47, 169]이 스트림 프로세서에 채택되고 있습니다. 이러한 추세는 올바른 물리적 계획을 선택하는 프로세스를 자동화하고 지속적인 애플리케이션이 실행되는 동안 해당 계획을 재구성할 수 있는 최적화 및 정교하면서도 투명한 상태 관리를 위한 새로운 기능을 열어줍니다.

## 5 내결합성

내결합성은 오류가 발생하지 않은 것처럼 작동을 계속하는 시스템의 능력입니다. 내결합성이 없으면 잠재적으로 제한되지 않은 데이터를 처리하는 스트리밍 시스템은 오류 중에 계산 상태가 손실된 경우 처음부터 데이터 처리를 반복해야 합니다. 더 나쁜 것은 입력 데이터 스트림이 오류 발생 후 복구되어 다시 처리될 수 있는지 여부가 불확실하다는 것입니다.

마지막으로, 분산 스트리밍 시스템 배포 규모는 내결합성의 중요성을 더욱 강조합니다.

규모가 클수록 오류가 시스템 작동에 더 많은 영향을 미칠 수 있습니다.

이 섹션은 오류 시나리오를 고려하여 스트리밍 시스템이 달성할 수 있는 정확성 수준을 특성화하는 처리 의미론에 대한 설명으로 시작합니다(섹션 5.1). 곧 총 5.2 내결합성에 상태 스냅샷의 중요한 역할을 설명합니다. 5.3 외부 세계에서 관찰되는 시스템 출력의 정확성을 고려하는 스트림 처리의 출력 커밋 문제. 섹션 5.4 에서는 내결합성과 밀접한 관련이 있는 스트리밍 시스템의 가용성에 관한 문헌을 제시합니다. 마지막으로 이 섹션은 내결합성(섹션 5.5) 및 공개 문제(섹션 5.6)와 관련하여 1세대 시스템과 2세대 시스템을 비교하는 것으로 끝납니다.

additional operation that is invoked prior to stateful processing that lifts the scope from task- to key-based processing such as “keyBy” in Apache Flink or “groupBy” in Beam and Kafka-Streams. Note that logical partitioning (i.e., which logical operator takes over a given key) differs from physical partitioning (i.e., which nodes take over the computations on a given set of keys). Typically, multiple keys (or key ranges) are assigned to a given compute node.

**Non-partitioned state** Non-partitioned state is mapped as a singleton to physical compute tasks. Such non-partitioned state is typically used in two ways. First, it can be used in order to compute global aggregates over the complete input stream. Second, it can be used to calculate aggregates at the level of the physical operator (e.g., count how many keys have been processed per operator). Task-level state can also be useful for keeping offsets when consuming logs from a physical stream source task. Because non-partitioned state either deals with operator-local computations or with global aggregates, its use is not scalable and should be used with caution by practitioners.

## 4.5 First versus second generation

State has been central to stream processing. The notion of state itself has been addressed with many names, such as “summary”, “synopsis”, “sketch” or “stream table” and it reflects the evolution of data stream management along the years. Early systems [12, 19, 27, 48] (circa 2000–2010) hid state and its management from the user. Most continuous processing operators at that time, such as those of the time-varying relational model of CQL [21] in STREAM [19], were implemented using internal in-memory data structures. Overall, the purpose of state was to support the creation of a limited set of operators offered by each system.

A decade later, scalable data computing systems based on the MapReduce [61] architecture allowed for arbitrary user-defined logic to be scaled and executed reliably using distributed middleware and partitioned file systems. Following the same trend, many existing data management models were revisited and re-architected with scalability in mind (e.g., NoSQL and NewSQL databases). Similarly, a growing number of scalable data stream processing systems [14, 16, 37, 118] married principles of scalable computing with stream semantics and models that were identified in the past (e.g. out-of-order processing [108, 142]).

As of today, modern stream processors can compile and execute graphs of long-running operators with complete, user-defined (yet system-managed) state that is fault-tolerant and reconfigurable given a clear set of transactional guarantees [14, 36, 64].

## 4.6 Open problems

Data streaming covers many data management needs today that go beyond real-time analytics, which was the original purpose of the stream processing technology. The transitions of stateful processing showcase this trend.

The decoupling of state programming from state persistence resembles the concept of data independence in databases. Systems are converging in terms of semantics and operations on state while, at the same time, many new methods employed on embedded databases (e.g., LSM-trees, state indexing, externalized state) are helping stream processors to evolve in terms of performance capabilities. Recent work [24, 88, 105] showcases the potential of workload-aware state management, adapting state persistence and access to the individual operators of a dataflow graph. To this end, an increasing number of “pluggable” systems [47, 169] for local state management with varying capabilities are being adopted by stream processors. This trend opens new capabilities for optimization and sophisticated, yet transparent, state management that can automate the process of selecting the right physical plan and reconfigure that plan while continuous applications are executed.

## 5 Fault tolerance

Fault tolerance is a system’s ability to continue its operation as if no failures have occurred. Without fault tolerance, streaming systems, which process potentially unbounded data, would have to repeat data processing from the beginning if the state of a computation was lost during a failure. Worse, it is uncertain whether an input data stream could be recovered following a failure, such that it can be processed again. Finally, the scale of distributed streaming system deployments highlights further the importance of fault tolerance. The larger the scale, the more a failure can affect the system’s operation.

We start this section with an account of processing semantics, which characterize the levels of correctness that a streaming system can achieve considering failure scenarios (Sect. 5.1). In Sect. 5.2 we describe the important role of state snapshots in fault tolerance. We devote Sect. 5.3 to the output-commit problem in stream processing, which regards the correctness of a system’s output as observed by the outside world. Section 5.4 presents the literature on the availability of streaming systems, which is closely related to fault tolerance. Finally, the section ends with a comparison between first and second generation systems with respect to fault tolerance (Sect. 5.5) and open problems (Sect. 5.6).

표 3 스트리밍 시스템의 내결함성

체계	처리 의미론		복제		복구 데이터		거래 세분성
	최소 상태	정확히 한 번 산출	활동적인	수동 없음 상태	출력 없음		
오로라* [52]						아니요	
텔레그램CQ [136]						아니요	
보레알리스 [11, 30]						아니요	
S4 [123]						아니요	
스며들다 [64, 65]						획기적인 수준	
나이아드 [120]						획기적인 수준	
타임스트림 [129]						획기적인 수준	
밀힐 [14]						기록적인 수준	
폭풍 [152]						아니요	
트라이언트 [9]						배치 수준	
S스토어 [42, 147]						배치 수준	
트릴 [44]						아니요	
헤론 [100]						아니요	
스트림스코프 [109]						획기적인 수준	
스트림 [56]						획기적인 수준	
삼자 [124]						획기적인 수준	
플링크 [36, 37]						획기적인 수준	
스파크 [23]						배치 수준	

## 5.1 처리 의미론

처리 의미론은 시스템 상태가 어떻게 영향을 받는지 전달합니다. 실패로. 일반적으로 문헌의 모든 시스템은 다음을 수행할 수 있습니다. 실패 없는 실행으로 올바른 결과를 생성합니다. 하지만 실패를 완전히 가리는 것은 어렵습니다. 특히 스트림에서는 더욱 그렇습니다. 출력이 다음과 같이 전달되어야 하는 처리 도메인 생산되자마자.

최근에는 스트림 처리 영역이 정착되었습니다. 특성화하기 위해 최소 한 번 및 정확히 한 번이라는 용어에 대해 처리 의미론 [23, 37, 56, 109, 124]. 최대 1회 명명법의 일부이기도 하지만 대부분 더 이상 사용되지 않습니다. 시스템은 두 가지 더 강력한 수준 중 하나를 지원하도록 선택합니다. ~에 최소 한 번 처리 의미론은 시스템이 실패 없는 실행과 동일한 결과를 생성합니다. 복구의 부작용으로 중복 레코드 추가. 우리는 다양한 시스템의 일관성 보장에 대해 자세히 설명합니다. 표 3.

정확히 한 번만 처리하면 두 가지 다른 해석이 가능해집니다. 시스템은 단 한 번의 처리를 지원할 수 있습니다. 복구 시 수행되는 불일치나 중복 실행이 발생하지 않도록 보장하는 경계 내의 의미 체계 상태의 일부입니다. 우리는 이 수준의 의미론을 정확히 한 번이라고 부릅니다. 상태에 대한 의미를 처리합니다. 이 범주에 속하는 대부분의 시스템은 여전히 그들이 적용하는 계산은 다음과 같다고 가정합니다.

시스템의 기능은 결정적이지만 종종 그렇지 않습니다.

경우; 처리 시간 창과 여러 소스의 입력을 처리하는 연산자는 두 가지 주 요 예입니다.

비결정론. 비결정론이 작용하면 시스템의 회복 상태는 다를 수 있습니다. Clanos [138]는 다음과 같은 비결정적 계산을 포함하여 정확히 한 번의 처리를 제공합니다. 인과적 일관성을 의미합니다. 그것은 결정 요인을 유지합니다 탄력적인 방식의 비결정론적 계산 및 사용 다음과 같은 정확한 계산 상태를 재생성합니다. 실패.

상태에 대해 정확히 한 번만 처리하는 의미 체계를 갖춘 시스템 여전히 중복 출력을 생성할 수 있습니다. 한 가지 주목할만한 예는 실패에서 복구하는 동안. 이런 결과가 가능합니다, 왜냐하면 시스템 상태와 반대로 출력은 롤백됩니다. 출력물을 즉시 소비될 수 있습니다

외부 응용 프로그램으로. 이 문제는 다음과 같이 명명되었습니다. 분산 시스템 문헌의 출력 커밋 문제 [62]. 장애가 발생하더라도 동일한 출력을 생성하는 시스템 오류 없는 실행 지원으로 정확히 한 번만 처리 가능 출력의 의미 . 곤충. 5.3 스트리밍 방법을 자세히 설명합니다. 시스템은 출력 커밋 문제를 해결합니다.

### 5.1.1 상태 일관성

일관된 스트리밍 처리는 어려운 성격으로 인해 꽤 오랫동안 공개 연구 문 제였습니다.

무제한 스트리밍의 분산 처리뿐만 아니라

**Table 3** Fault-tolerance in streaming systems

System	Processing semantics		Replication		Recovery data			Transaction granularity
	Least State	Exactly-once Output	Active	Passive	None	State	Output	None
Aurora* [52]	✓			✓		✓		No
TelegraphCQ [136]		✓		✓		✓	✓	No
Borealis [11, 30]	✓			✓		✓	✓	No
S4 [123]	✓				✓		✓	No
Seep [64, 65]		✓		✓		✓	✓	Epoch-level
Naiad [120]		✓		✓		✓	✓	Epoch-level
Timestream [129]		✓		✓		✓	✓	Epoch-level
Millwheel [14]		✓		✓		✓	✓	Record-level
Storm [152]	✓				✓		✓	No
Trident [9]		✓		✓		✓		Batch-level
S-Store [42, 147]	✓			✓		✓		Batch-level
Trill [44]	✓			✓		✓		No
Heron [100]	✓				✓		✓	No
Streamscope [109]		✓	✓	✓	✓	✓	✓	Epoch-level
Streams [56]	✓			✓		✓		Epoch-level
Samza [124]	✓			✓		✓		Epoch-level
Flink [36, 37]	✓			✓		✓		Epoch-level
Spark [23]	✓			✓		✓		Batch-level

## 5.1 Processing semantics

Processing semantics conveys how a system’s state is affected by failures. Typically, all systems in the literature are able to produce correct results in failure-free executions. But to mask a failure completely is hard, especially in the stream processing domain where the output should be delivered as soon as it is produced.

In recent years, the stream processing domain has settled on the terms *at least-once* and *exactly-once* to characterize the processing semantics [23, 37, 56, 109, 124]. At most-once is also part of the nomenclature but it is mostly obsolete, as systems opt to support one of the two stronger levels. At least-once processing semantics means that the system will produce the same results as a failure-free execution with the addition of duplicate records as a side-effect of recovery. We detail the consistency guarantees of different systems in Table 3.

**Exactly-once processing** lends itself to two different interpretations. A system may support exactly-once processing semantics within its boundaries ensuring that any inconsistencies or duplicate execution carried out on recovery is not part of its state. We call this level of semantics exactly-once processing semantics on *state*.

It should be noted that most systems in this category still assume that the computations they apply, as well as

the system’s functions, are deterministic, which is often not the case; processing-time windows and operators processing input from multiple sources are two prime examples of nondeterminism. With nondeterminism at play, the system’s state on recovery can diverge. Clanos [138] provides exactly-once processing including nondeterministic computations by means of causal consistency. It keeps determinants about nondeterministic computations in a resilient manner and uses them to regenerate the exact computational state following a failure.

A system with exactly-once processing semantics on state can still produce duplicate output. One notable example is while recovering from a failure. This outcome is possible, because, as opposed to the system’s state, the output cannot be rolled back. The output can be consumed immediately by external applications. This problem has been termed the *output-commit problem* [62] in the distributed-systems literature. Systems that produce the same output under failure as a failure-free execution support exactly-once processing semantics on *output*. In Sect. 5.3 we elaborate how streaming systems address the output-commit problem.

### 5.1.1 State consistency

Consistent stream processing has been an open research problem for quite some time due to the challenging nature of distributed processing of unbounded streams, but also due to

문제 자체에 대한 공식적인 정의가 부족합니다. 일관성은 오류 발생 시 시스템이 제공할 수 있는 보장과 관련이 있습니다.

람다 아키텍처 클라우드 컴퓨팅의 출현과 함께 "람다 아키텍처"라는 디자인 패턴이 주류가 되었습니다. 람다 아키텍처는 전문화 및 안정성 기능에 따라 여러 계층에 걸쳐 시스템을 분리하는 것을 제안했습니다. Hadoop은 처리 보장(즉, 일괄 데이터를 원자적으로 처리하여 정확히 한 번 처리) 측면에서 신뢰할 수 있으므로 올바른 계산을 실행할 수 있었습니다. 그러나 Hadoop 기본 솔루션은 대기 시간이 길어 어려움을 겪었습니다. 반면, 초기 스트림 처리 시스템은 낮은 대기 시간을 달성할 수 있었지만 일관성 보장을 제공하지 않았습니다. 대부분 최소 한 번 의미론을 보장했습니다.

동시에 데이터베이스에는 공식적인 보증이 있었습니다. 예를 들어 일련의 거래는 ACID 보장을 사용하여 처리됩니다. 그러나 데이터 스트리밍의 맥락에서 정확히 1회 처리의 필요성과 가능한 재생을 위한 입력 로깅의 필요성을 결정하는 데 시간이 좀 걸렸습니다. 다음에서는 처리 보장에 대해 논의합니다.

스트림 처리의 일관된 상태 오늘날의 스트림 프로세서는 동시에 실행되는 다양한 작업으로 구성된 분산 시스템입니다. 소스 작업은 일반적으로 Kafka와 같이 분할된 로그에 기록되는 입력 스트림을 구독하므로 입력 스트림을 재생할 수 있습니다. 싱크 작업은 출력 스트림을 외부 세계에 커밋합니다. 이 시스템의 모든 작업은 자체 상태를 포함할 수 있습니다. 예를 들어 소스 작업은 입력 스트림의 현재 위치를 해당 상태로 유지해야 합니다. 시스템 실행은 종종 "동시 작업" 개념을 통해 모델링될 수 있습니다 [35]. 작업에는 입력 이벤트에 대한 스트림 작업 논리 호출, 해당 상태 변경 및 출력 이벤트 생성이 포함됩니다. 이러한 시스템에서 발생하는 모든 작업은 다른 작업을 유발합니다. 실제로 소스에서 보낸 단일 레코드만 전체 파이프라인의 상태 업데이트와 싱크에서 생성된 출력 이벤트에 기여합니다. 특정 작업이 손실되거나 두 번 이상 발생하면 전체 시스템이 일관성 없는 상태가 됩니다.

내결합성은 상태 일관성에 큰 영향을 미치는 스트리밍 시스템의 필수적인 측면입니다. Sect.에서는 기존 스트리밍 시스템의 내결합성 전략을 분석합니다. 5.1.2. 또한 상태에 대한 인과적 종속성으로 인해 작업 실행 순서도 중요합니다. 원인은 다음과 같습니다.

여기서 중요한 것은 이벤트가 시스템에 입력된 후 작업 동작의 실행 순서에만 관련되어 이벤트가 수집되는 순서에는 적용되지 않습니다. 따라서 시스템은 이벤트 타임스탬프와 관련하여 순서가 잘못된 스트림을 허용할 수 있습니다. 그러나 이는 처리된 이벤트 스트림 간의 데이터 흐름 실행 내에서 인과 관계 종속성을 유지할 수 있다는 의미는 아닙니다(수집 순서에 상관없이). 기준의

신뢰할 수 있는 스트림 프로세서는 각 작업에서 트랜잭션을 정의하거나 에포크라고 부르는 대략적인 작업 집합을 정의합니다. 다음에서 이러한 접근 방식에 대해 더 자세히 설명합니다.

### 5.1.2 일관성 메커니즘의 속성

분산 스트리밍 시스템의 장애 관리에는 상태 스냅샷 유지, 상태 마이그레이션, 운영자 확장이 수반되지만 시스템의 정상 부분에는 가능한 한 적은 영향을 미칩니다. 표 3은 과거부터 현재까지 출판을 출현 순서대로 정렬된 알려진 스트리밍 시스템의 내결합성 전략을 나타냅니다. 우리는 다음 세 가지 차원에 걸쳐 전략을 분석합니다.

1. 복제는 실행 복구를 위해 추가적인 계산 자원의 사용을 고려합니다. 우리는 Hwang et al.의 용어를 채택합니다. [85] 이는 복제를 동일한 실행의 두 인스턴스가 병렬로 실행되는 활성 또는 실행 중인 각 상태 저장 연산자가 체크포인트 상태를 대기 연산자로 전달하는 수동으로 분류합니다.

2. 복구 데이터는 복구 목적으로 정기적으로 저장되는 데이터를 지정합니다. 데이터에는 각 연산자의 상태 와 해당 연산자가 생성하는 출력이 포함될 수 있습니다. 또한 많은 내결합성 전략에서는 입력 스트림을 재처리하기 위해 복구 중에 입력 스트림의 튜플을 재생해야 합니다. 이를 위해 입력 스트림은 일반적으로 Apache Kafka와 같은 메시지 브로커에 지속적으로 저장됩니다. 스트림 처리 소비자 작업은 복구를 위해 해당 상태 내에서 입력 로그의 읽기 위치만 저장하면 됩니다.

3. 트랜잭션 세분성은 상태 스냅샷을 획득하는 빈도에 따라 시스템을 기록, 에포크 또는 배치 수준 스냅샷 빈도로 분류하는 것을 고려합니다.

이 표는 내결합성에 대한 특정 시스템의 접근 방식을 설명하기 위해 가로로 읽고, 스트림 처리에서 다양한 구성 요소가 내결합성의 환경을 형성하는 방법을 밝히기 위해 세로로 읽어야 합니다. 두 가지 언급이 필요합니다. Streamscope [109]는 세 가지 별개의 내결합성 전략을 제시하고 평가합니다. 활성 복제 기반 전략, 수동적 전략 및 입력 스트림에서 데이터를 재생하여 상태를 다시 계산하는 전략입니다. 또한 복구 데이터 차원의 상태 열은 체크포인트 상태뿐만 아니라 변경 로그 [124] 또는 상태 증속성 [129]과 같은 상태를 다시 계산할 수 있는 상태 메타데이터도 캡처합니다.

표에는 네 가지 흥미로운 패턴이 나와 있습니다. 첫째, 모든 열 중에서 수동 복제와 복구를 위한 상태 저장이라는 두 열에 대부분의 확인 표시가 촉적됩니다. 이 패턴은 아마도 표에서 가장 눈에 띄는 패턴일 것이며 상태 저장에 의한 수동 복제가 스트리밍 시스템에서 매우 인기 있는 옵션이라는 것을 의미합니다. 일반적인 복구 접근 방식 중 하나는 새 노드에서 실패한 연산자의 최신 체크포인트를 복원하고 체크포인트 이후에 나타난 입력을 재생하는 것입니다. 이 접근 방식의 변형에는 다음이 포함됩니다.

the lack of a formal definition of the problem itself. Consistency relates to the guarantees that a system can provide in face of failures.

**The Lambda architecture** With the advent of cloud computing, a design pattern called the “lambda architecture” became mainstream. The lambda architecture proposed the separation of systems across different layers according to their specialization and reliability capabilities. Hadoop was reliable in terms of processing guarantees (i.e., exactly-once processing by atomically processing batches of data), thus, it could execute correct computation. However, Hadoop-based solutions suffered from high latency. On the other hand, early stream-processing systems could achieve low latency but they did not offer consistency guarantees—they mostly guaranteed at-least-once semantics.

At the same time, databases had formal guarantees. For example, a set of transactions would be processed using ACID guarantees. In the context of data streaming, though, it took some time in order to decide on the need for exactly-once processing, and the need for input logging for a possible replay. We discuss processing guarantees in the following.

**Consistent state in stream processing** A stream processor today is a distributed system consisting of different concurrently executing tasks. Source tasks subscribe to input streams that are typically recorded in a partitioned log, such as Kafka, and therefore, input streams can be replayed. Sink tasks commit output streams to the outside world. Every task in this system can contain its own state. For example, source tasks need to keep the current position of their input streams in their state. A system execution can be often modeled through the concept of “concurrent actions” [35]. An action includes: invoking stream task logic on an input event, mutating its state, and producing output events. Every action happening in such a system causes other actions. Effectively, just a single record sent by a source contributes to state updates throughout the whole pipeline and to output events created by the sinks. If a specific action is lost or happens more than once, then the complete system enters into an *inconsistent* state.

Fault tolerance is an integral aspect of streaming systems that significantly affects their state consistency. We analyze the fault tolerance strategies of existing streaming systems in Sect. 5.1.2. In addition, due to causal dependencies on state, the order of action execution is also critical. Note that causality here concerns only the execution order of task actions after events enter the system, not the order in which events are ingested. Therefore, a system can tolerate out-of-order streams with respect to event timestamps. However, this does not mean that it is capable of maintaining causal dependencies within its dataflow execution among the event streams that have been processed (in either ingestion order). Existing

reliable stream processors either define a transaction out of each action or a coarse-grained set of actions that we call *epochs*. We explain these approaches in more detail, next.

### 5.1.2 Properties of consistency mechanisms

Managing failures in a distributed streaming system entails maintaining snapshots of state, migrating state, and scaling out operators while affecting the healthy parts of the system as little as possible. Table 3 presents the fault-tolerance strategies of known streaming systems arranged in order of publication appearance, from past to present. We analyze the strategies across the following three dimensions.

1. *Replication* considers the use of additional computational resources for recovering an execution. We adopt the terminology of Hwang et al. [85] that classifies replication as either *active*, where two instances of the same execution run in parallel, or *passive*, where each running stateful operator dispatches its checkpointed state to a standby operator.

2. *Recovery data* addresses what data are regularly stored for recovery purposes. Data may include the *state* of each operator and the *output* it produces. In addition, many fault tolerance strategies need to replay tuples of input streams during recovery in order to reprocess them. For this purpose, input streams are persistently stored, typically in message brokers like Apache Kafka. Stream processing consumer tasks only need to store read input positions from input logs within their state for recovery.

3. *Transaction Granularity* considers the categorization of systems by the frequency at which they obtain snapshots of their state into Record-, Epoch- or Batch-level snapshot frequency.

The table is meant to be read both horizontally, to describe a specific system’s approach to fault tolerance, and vertically, to uncover how the different building blocks shape the landscape of fault tolerance in stream processing. Two remarks are necessary. Streamscope [109] presents and evaluates three distinct fault-tolerance strategies; an active replication-based strategy, a passive one, and a strategy that relies on recomputing state by replaying data from input streams. Furthermore, the state column in the recovery data dimension captures not only checkpointed state but also state metadata that allow recomputing the state, such as a changelog [124] or state dependencies [129].

The table reveals four interesting patterns. First, of all columns, two accumulate the majority of checkmarks: passive replication and storing state for recovery. This pattern is perhaps the most visible on the table and signifies that passive replication by storing state is, unsurprisingly, a very popular option for streaming systems. One typical recovery approach is to restore the latest checkpoint of a failed operator in a new node and replay input that appeared after the checkpoint. Variations of this approach include saving in-

상태와 함께 비행 투플을 유지하고 업스트림 노드에서 비행 중 투플을 유지합니다. 둘째, 복구를 위해 내부 투플을 저장하는 것은 과거에 스트리밍 시스템에서 인기 있는 옵션이었지만 더 이상 선호되지 않습니다. 이러한 변화를 설명하는 주요 개발 중 하나는 Apache Kafka [98]와 같은 메시지 브로커의 출현입니다. 이는 오프셋을 사용하여 특정 시점부터 시작하여 정렬된 데이터 스트림을 생성할 수 있습니다. 복구 시 스트리밍 시스템을 통해. 셋째, 과거 시스템은 정확히 1회 출력 처리 의미 체계를 지원하려고 노력했지만 이후 시스템은 상태에 대해 정확히 1회 의미 체계를 선택하고 출력 중복 제거를 외부 시스템에 아웃소싱합니다. 이 부분에 대해서는 섹션에서 자세히 설명하겠습니다. 5.3.

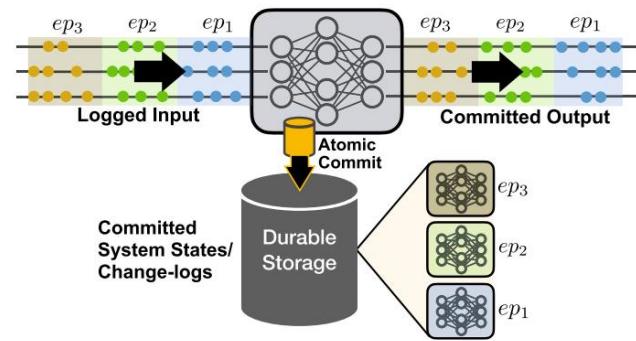


그림 4 데이터 스트리밍의 트랜잭션 에포크 커밋

## 5.2 내결합성 및 상태 스냅샷

스냅샷은 오류 복구 및 재구성을 목적으로 하는 스트림 처리 상태의 지속적인 복사본입니다. 각 시스템의 일관성 속성과 사용되는 메커니즘은 해당 복사본을 얼마나 자주 얻어야 하는지에 대해 서로 다른 요구 사항을 부과합니다. 우리는 상태 스냅샷을 기록 및 에포크 수준 스냅샷 빈도로 기록하는 빈도에 따라 시스템을 차별화합니다.

레코드 세분성 시대 기반 접근 방식의 반대 극단은 각 레코드 이후의 스냅샷을 만드는 것입니다. Millwheel [14]에서 볼 수 있듯이 이 접근 방식에서는 스트림 프로세서가 각 스트림 레코드의 처리가 완료된 후 상태에서 발생한 모든 변경 사항의 복사본을 얻어야 합니다. 이 복사본에는 새로 업데이트된 상태와 생성된 출력 레코드가 포함됩니다.

Epoch 세분성은 일반적으로 애플리케이션 수준 스냅샷의 형태로 달성됩니다. 가장 일반적으로 시스템은 Chandy-Lamport 알고리즘 [49]과 같은 비동기식 일관된 스냅샷 형식을 사용합니다. 각 시대마다, 즉 주기적으로 또는 특정 수의 레코드가 시스템에 수집된 후에 각 운영자는 해당 상태의 복사본을 기록합니다. Spark Streaming 및 Trident/Storm과 같은 시스템에서 볼 수 있는 배치 수준 스냅샷은 엄격한 마이크로 배치 처리 패러다임을 채택합니다. 즉, 상당한 수의 레코드를 수집한 후 배치 실행이 제출되고 그 직후에 운영자의 상태가 저장됩니다. 특정 배치가 처리되었습니다. S-Store는 관계형 데이터베이스 위에 일련의 ACID 트랜잭션으로 배치 세분성을 조정합니다.

### 5.2.1 기록 세분성에서의 상태 내구성

데이터 스트리밍의 일관된 처리 형태는 로컬 작업당 트랜잭션을 사용하는 것입니다. 데이터 흐름 데이터 스트리밍 서비스를 위한 클라우드 런타임인 Google의 Millwheel은 이러한 전략을 사용합니다. Millwheel은 BigTable을 사용하여 입력 이벤트, 상태를 포함하는 각 전체 컴퓨팅 작업을 커밋합니다.

전환 및 생성된 출력. 이러한 행위를 저지르는 행위를 밀휠에서는 '강력한 생산'이라고도 합니다.

출력 이벤트당 연산자의 상태를 유지하는 것은 대기 시간이 긴 오버헤드를 유발하는 것처럼 보이는 접근 방식입니다. 그러나 기존 데이터베이스 최적화를 사용하여 커밋 및 상태 읽기 시간을 단축할 수 있습니다. 미리 쓰기 로깅, 블라인드 쓰기, 블룸 필터 및 스토리지 계층의 일괄 커밋을 사용하여 커밋 대기 시간을 줄일 수 있습니다. 더 중요한 점은 작업 순서가 커밋 시 미리 정의되어 있기 때문에 이벤트별 상태 지속성이 결정적 실행도 보장한다는 것입니다. 또한 이 접근 방식은 시스템의 출력을 소비하는 응용 프로그램이 인식하는 일관성에 중요한 영향을 미칩니다. 이러한 이점은 이 맥락에서 "정확히 한 번 처리"가 각 작업이 원자 단위로 커밋되는 것과 관련이 있다는 사실에서 비롯됩니다. 5.3.

이 접근 방식의 정확성 뒤에 있는 핵심 관찰은 각 작업이 원자 단위로 커밋된 작업에 의해 호출된다는 사실에 기반합니다. 따라서 작업의 자연스러운 인과 순서에 따라 시스템의 모든 작업이 전이적으로 종료되는 것도 원자적으로 커밋됩니다.

### 5.2.2 에포크 세분성에서의 상태 내구성

에포크 수준 처리에서 상태당 기록 빈도는 레코드별 세분성에 비해 더 대량입니다.

그림 4에서는 입력, 시스템 상태 및 출력을 고유한 에포크 식별자로 표시하는 전반적인 접근 방식을 설명합니다.

스트리밍 애플리케이션의 기록된 입력에 있는 마커를 통해 에포크를 정의할 수 있습니다. 각 시대를 처리하고 각 시대가 처리된 후 전체 작업 그래프의 상태를 커밋하도록 시스템 실행을 계획할 수 있습니다. epoch 실행 중에 오류나 기타 재구성 작업이 발생하면 시스템은 이전에 커밋된 epoch로 롤백하고 실행을 복구할 수 있습니다.

이 맥락에서 "정확히 한 번만 처리"라는 용어는 원자적으로 커밋되는 각 시대와 관련이 있습니다. 곤충. 5.1.2에서는 스트리밍의 다양한 수준의 처리 의미를 제시하면서 이 플레이어를 정확히 한 번만 처리한다고 불렀습니다.

flight tuples along with the state and maintaining in-flight tuples in upstream nodes. Second, storing in-flight tuples for recovery is not preferred anymore, although it was a popular option for streaming systems in the past. One major development that explains this shift is the advent of message brokers, such as Apache Kafka [98], which can produce an ordered stream of data starting from a specific point in time using an offset, which is, for instance, provided by a streaming system on recovery. Third, while past systems strived to support exactly-once output processing semantics, later systems opt for exactly-once semantics on state and outsource the deduplication of output to external systems. We will elaborate on this aspect in Sect. 5.3.

## 5.2 Fault tolerance and state snapshots

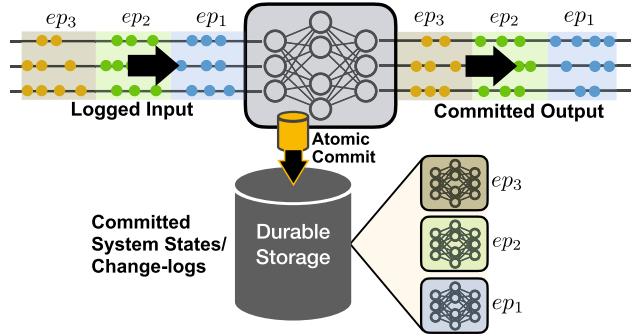
Snapshots are persisted copies of the stream-processing states for the purposes of fault recovery and reconfiguration. The consistency properties and employed mechanisms in each system impose different requirements on how often such copies need to be obtained. We differentiate systems by the frequency at which they record snapshots of their state into Record- and Epoch-level snapshot frequency.

**Record-granularity** An opposite extreme to the epoch-based approach is to snapshot after each record. This approach, as seen in Millwheel [14], requires the stream processor to obtain a copy of all changes that occurred in the state after the complete processing of a each stream record. This copy includes the newly updated states and produced output records.

**Epoch-granularity** is typically achieved in the form of application-level snapshots. Most commonly, systems employ a form of asynchronous consistent snapshotting, such as the Chandy–Lamport algorithm [49]. In each epoch, i.e., either periodically or after a certain number of records have been ingested by the system, each operator records a copy of its state. The batch-level snapshotting seen in systems such as Spark Streaming and Trident/Storm adopts a strict micro-batching processing paradigm: i.e., a batch execution is submitted after collecting a sizable number of records, and the state of an operator is stored right after a given batch has been processed. S-Store orchestrates the batch granularity as a series of ACID transactions on top of a relational database.

### 5.2.1 State durability at record granularity

A form of consistent processing in data streaming is employing a transaction per local action. Google’s Millwheel, the cloud runtime for the dataflow data streaming service, employs such a strategy. Millwheel uses BigTable to commit each full compute action which includes: input events, state



**Fig. 4** Transactional epoch commits in data streaming

transitions and generated output. The act of committing these actions is also called a “strong production” in Millwheel.

Persisting state of an operator per output event, is an approach which seemingly induces high latency overhead. However, traditional database optimizations can be used to speed up commit and state read times. Write ahead logging, blind writes, bloom filters, and batch commits at the storage layer can be used to reduce the commit latency. More importantly, since the order of actions is predefined at commit time, state persistence on a per-event basis also guarantees deterministic executions. In addition, this approach has important effects on consistency as perceived by applications that consume the system’s output. This benefit follows from the fact that “exactly-once processing” in this context relates to each action being atomically committed, as we discuss in Sect. 5.3. The core observation behind the correctness of this approach is based on the fact that each action is invoked by an atomically committed action. Following the natural causal order of actions, the transitive closure of all actions in the system is, therefore, also atomically committed.

### 5.2.2 State durability at epoch granularity

The frequency of recordings per state in epoch-level processing is more coarse-grained compared to the per-record granularity.

In Fig. 4 we depict the overall approach, marking input, system states, and outputs with a distinct epoch identifier. Epochs can be defined through markers at the logged input of the streaming application. A system execution can be instrumented to process each epoch and commit the state of the entire task graph after each epoch is processed. If a failure or other reconfiguration action happens during the execution of an epoch, then the system can roll back to a previously committed epoch and recover its execution. The term “exactly-once processing” in this context relates to each epoch being atomically committed. In Sect. 5.1.2, where we presented the different levels of processing semantics in streaming, we called this flavor *exactly-once processing* on

## 스트림의 진화에 관한 조사…

상태. 이 섹션의 나머지 부분에서는 스트림 시대를 커밋하는 데 사용되는 다양한 접근 방식에 중점을 둡니다.

엄격한 2단계 에포크 커밋 에포크를 커밋하기 위한 일반적인 조정 프로토콜은 1단계가 에포크의 전체 처리에 해당하고 2단계에서 계산이 끝날 때 시스템 상태를 유지하는 엄격한 2단계 커밋입니다.

이 접근 방식은 주기적인 "マイクロ 배치"를 사용하여 Apache Spark [164]에 의해 대중화되었으며 무제한 처리를 위해 배치 처리 시스템을 사용할 때 효과적인 전략입니다. 이 접근 방식의 주요 단점은 동기 실행으로 인해 작업 활용도가 낮아질 위험이 있다는 것입니다. 각 작업은 다른 모든 작업이 현재 에포크를 완료할 때까지 기다려야 하기 때문입니다. Drizzle [158]은 단일 원자 커밋에서 여러 시대를 연결하여 이 문제를 완화합니다. S-Store는 유사한 접근 방식 [116]을 사용합니다. 여기서 각 데이터베이스 트랜잭션은 이미 동일한 데이터베이스에 저장된 입력 스트림의 시대에 해당합니다.

비동기적 2단계 에포크 커밋 순수 데이터 흐름 시스템의 경우 엄격한 2단계 커밋은 문제가 됩니다. 작업이 조정되지 않고 오래 실행되기 때문입니다. 또한, 고전적인 분산 시스템 문헌 [35]에서 알려진 일관된 스냅샷 알고리즘을 통해 동일한 기능을 비동기적으로 달성하는 것이 가능합니다. 일관된 스냅샷 알고리즘은 스트리밍 애플리케이션을 일시 중지할 필요가 없기 때문에 유용한 속성을 나타냅니다. 또한 분산 실행에서 일관된 컷의 스냅샷을 획득합니다 [49]. 즉, "유효한" 실행 중에 시스템의 전역 상태를 캡처합니다. 다양한 구현을 통해 i) 정렬되지 않은 프로토콜과 ii) 정렬된 스냅샷 프로토콜을 식별할 수 있습니다.

I. 정렬되지 않은(Chandy-Lamport [49]) 스냅샷은 일관된 결과를 얻는 가장 효율적인 방법 중 하나를 제공합니다.

스냅 사진. 현재 IBM Streams와 같은 여러 스트림 프로세서가 이 접근 방식을 지원하고 있으며 최근에는 Flink [10]에서 실험적 구성 옵션으로 나타났습니다. 핵심 아이디어는 정기적인 이벤트 스트림에 구두점이나 "마커"를 삽입하고 해당 마커를 사용하여 시스템이 실행되는 동안 스냅샷 전후의 모든 작업을 분리하는 것입니다. 정렬되지 않은 스냅샷의 주의 사항은 프로토콜이 완료될 때까지 개별 작업에 도착하는 입력(즉, 진행 중) 이벤트를 기록해야 한다는 것입니다. 기록된 입력에 대한 공간 오버헤드 외에도 정렬되지 않은 스냅샷은 기록된 입력을 재생해야 하기 때문에 복구 중에 더 많은 처리가 필요합니다(비슷한 검사점을 사용하여 데이터베이스 복구에서 로그를 다시 실행하는 것과 유사).

II. 정렬된 스냅샷 정렬된 스냅샷은 복구 중 성능을 향상시키고 정렬되지 않은 스냅샷에서 나타나는 재구성 복잡성을 최소화하는 것을 목표로 합니다. 주요 차이점은 예상되는 입력 스트림 이벤트의 우선순위를 정한다는 것입니다.

따라서 스냅샷의 일부로 진행 중인 이벤트가 없고 에포크의 완전한 계산을 반영하는 상태로만 끝나게 됩니다. 예를 들어, Flink의 시대 스냅샷 메커니즘 [36, 38]은 마커를 사용하여 시대 경계선을 식별한다는 측면에서 Chandy Lamport 알고리즘과 유사합니다. 그러나 추가로 전파하기 전에 작업 내에서 마커를 동기화하는 정렬 단계도 사용합니다. 정렬은 모든 입력 채널이 특정 에포크에 해당하는 모든 메시지를 전송할 때까지 이전에 마커가 수신되었던 입력 채널을 부분적으로 차단함으로써 달성됩니다.

요약하면 정렬되지 않은 스냅샷은 최고의 런타임 성능을 제공하지만 복구 시 필요한 다시 실행 단계로 인해 복구 시간이 희생됩니다. 반면, 정렬된 스냅샷은 정렬 단계로 인해 커밋 시간이 느려지는 동시에 일련의 유익한 속성을 제공할 수 있습니다.

첫째, 정렬된 스냅샷은 한 시대의 완전한 실행을 반영하며, 이는 데이터 스트리밍 외에 스냅샷 격리 쿼리를 지원해야 하는 사용 사례에 유용합니다 [159].

또한 정렬된 스냅샷은 재구성 공간이 가장 낮을 뿐만 아니라 Chi [112]에서 설명한 것처럼 정렬 단계 내에서 실시간 재구성을 위한 기반을 설정합니다.

## 5.3 출력 커밋 문제

출력-커밋 문제 [62]는 시스템이 출력이 게시된 위치에서 상태를 복구할 수 있다는 확신 하에 시스템이 출력을 외부 세계에만 게시해야 함을 지정합니다. 이는 모든 출력이 한 번만 게시되도록 하기 위한 것입니다. 출력이 외부 세계에 공개되면 철회될 수 없기 때문입니다. 출력이 두 번 게시되면 시스템은 외부 세계와 관련하여 일관되지 않은 동작을 나타냅니다. 외부 세계를 구성하는 시스템은 문제의 범위를 벗어나므로 실패할 수 없다고 가정됩니다. 이 문제의 중요한 사례는 시스템이 오류로 인해 이전의 일관된 상태를 복원할 때 나타납니다.

출력 커밋 문제는 일반적으로 분산 아키텍처를 따르고 제한되지 않은 데이터 스트림을 처리하는 스트리밍 시스템과 관련이 있습니다. 이 설정에서는 실패의 부작용을 가리기가 어렵습니다. 예를 들어 상태에 대해 정확히 한 번만 처리하는 의미체계를 갖춘 스트리밍 시스템이 스냅샷을 찍고 얼마 지나지 않아 해당 연산자 중 하나가 충돌한다고 가정해 보겠습니다. 운영자는 해당 상태의 스냅샷을 찍은 후 충돌이 발생할 때까지 계속해서 출력을 생성했습니다. 운영자가 가장 최근 스냅샷을 사용하여 복구하면 스냅샷 작업에 성공한 입력 데이터를 다시 처리합니다. 결과적으로 마지막 스냅샷을 찍은 후 충돌이 발생하기 전에 정상적인 작동 하에서 생성한 출력을 다시 생성합니다.

출력 커밋 문제를 해결하는 스트리밍 시스템은 (1) 정확히 한 번 처리 의미론을 제공하는 것을 포함하여 문헌에서 여러 설명을 받았습니다.

*state.* The rest of this section focuses on various approaches used to commit stream epochs.

**Strict two-phase epoch commits** A common coordinated protocol to commit epochs is a strict two-phase commit where Phase 1 corresponds to the full processing of an epoch and Phase 2 ensures persisting the state of the system at the end of the computation.

This approach was popularized by Apache Spark [164] through the use of periodic “micro-batching” and it is an effective strategy when using batch processing systems for unbounded processing. The main downside of this approach is the risk of low task utilization due to synchronous execution, since each task has to wait for all other tasks to finish their current epoch. Drizzle [158] mitigates this problem by chaining multiple epochs in a single atomic commit. S-Store employs a similar approach [116], where each database transaction corresponds to an epoch of the input stream that is already stored in the same database.

**Asynchronous two-phase epoch commits** For pure dataflow systems, strict two-phase committing is problematic, since tasks are uncoordinated and long-running. Furthermore, it is feasible to achieve the same functionality asynchronously through consistent snapshotting algorithms, known from classic distributed systems literature [35]. Consistent snapshotting algorithms exhibit beneficial properties because they do not require pausing a streaming application. Furthermore, they acquire a snapshot of a consistent cut in a distributed execution [49]. In other words, they capture the global states of the system during a “valid” execution. Throughout different implementations we can identify i) unaligned and ii) aligned snapshotting protocols.

**I. Unaligned (Chandy–Lamport [49])** snapshots provide one of the most efficient methods to obtain a consistent snapshot. Several stream processors currently support this approach, such as IBM Streams and more recently seen as an experimental configuration option in Flink [10]. The core idea is to insert a punctuation or “marker” into the regular stream of events and use that marker to separate all actions that come before and after the snapshot, while the system is running. A caveat of unaligned snapshots is the need to record input (a.k.a. in-flight) events that arrive to individual tasks until the protocol is complete. In addition to space overhead for logged inputs, unaligned snapshots require more processing during recovery, since logged inputs need to be replayed (similarly to redo logs in database recovery with fuzzy checkpoints).

**II. Aligned snapshots** Aligned snapshots aim to improve performance during recovery and minimize reconfiguration complexity exhibited by unaligned snapshots. The main difference is prioritizing input stream events that are expected

before the snapshot and, thus, end up solely with states that reflect a complete computation of an epoch and no in-flight events as part of a snapshot. For example, Flink’s epoch snapshotting mechanism [36, 38] resembles the Chandy Lamport algorithm in terms of using markers to identify epoch frontiers. However, it additionally employs an alignment phase that synchronizes markers within tasks before disseminating them further. Alignment is achieved through partially blocking input channels where markers were previously received until all input channels have transferred all messages corresponding to a particular epoch.

In summary, unaligned snapshots are meant to offer the best runtime performance but sacrifice recovery times due to the redo-phase needed upon recovery. Whereas, aligned snapshots can lead to slower commit times due to the alignment phase, while providing a set of beneficial properties. First, aligned snapshots reflect a complete execution of an epoch, which is useful in use cases where snapshot-isolated queries need to be supported on top of data streaming [159]. Furthermore, aligned snapshots yield the lowest reconfiguration footprint, as well as set the basis for live reconfiguration within the alignment phase as exhibited by Chi [112].

### 5.3 The output-commit problem

The output-commit problem [62] specifies that a system should only publish its output to the outside world, under the certainty that the system can recover its state from where the output was published. This is to ensure that every output is only published once, since output cannot be retracted once it is made available to the outside world. If output is published twice, then the system manifests inconsistent behavior with respect to the outside world. The systems comprising the outside world fall out of the problem’s scope and, thus, it is assumed that they cannot fail. An important instance of this problem manifests when a system is restoring some previous consistent state due to a failure.

The output-commit problem is relevant in streaming systems, which typically conform to a distributed architecture and process unbounded data streams. In this setting, the side effects of failures are difficult to mask. For example, assume that a streaming system with exactly-once processing semantics on state takes a snapshot and, shortly afterward, one of its operators crashes. After the operator took a snapshot of its state, it continued to produce output until it crashed. When the operator recovers using the most recent snapshot, it will process the input data that succeeded the snapshot operation again. Consequently, it will re-produce the output that it had produced under its normal operation after taking the last snapshot and before suffering the crash.

Streaming systems that solve the output-commit problem have received multiple descriptions in the literature, including that they provide (1) exactly-once processing semantics

출력에 있어서, (2) 정확히 한 번만 출력하고, (3) 정확한 회복 [85], (4) 강력한 생산 [14].

문제가 관련성이 높고 어렵지만 스트림 처리 영역의 솔루션은 각 시스템과 관련된 문헌에 개별적으로 분산되어 있습니다. 우리는 다양한 솔루션을 트랜잭션 기반, 진행 기반, 계보 기반의 세 가지 범주로 분류합니다. 이제 우리는 관련된 가정에 초점을 맞춰 각각을 설명합니다. 세 가지 유형의 기술은 각각 입력 또는 계산의 서로 다른 특성을 사용하여 특정 템플릿이 이전에 나타났는지 여부를 식별합니다. 트랜잭션 기반 기술은 템플릿 ID를 사용하는 반면 진행 기반 기술은 순서를 사용합니다.

마지막으로 계보 기반 기술은 입력-출력 종속성을 사용합니다. 마지막으로 우리는 실제로 문제를 해결하는 특수 싱크 연산자와 외부 싱크라는 두 가지 범주의 솔루션을 더 제공하지 않습니다. 염밀히 말하면 스트리밍 시스템에 특정하거나 외부에 있기 때문에 문제의 사양을 충족하지 않습니다.

트랜잭션 기반 Millwheel [14] 및 Trident [9]는 중복 재시도를 제거하기 위해 레코드가 있는 고유 ID 커밋에 의존합니다. Millwheel은 시스템에 입력되는 각 레코드에 고유 ID를 할당하고 생성된 모든 레코드를 다운스트림으로 보내기 전에 고가용성 스토리지 시스템에 커밋합니다.

다운스트림 운영자는 수신된 레코드를 확인합니다. Mill-wheel은 입력 순서나 결정론을 가정하지 않습니다. 반면에 Trident는 고유한 트랜잭션 ID가 할당되고 상태 업데이트를 상태 백엔드에 적용하는 트랜잭션으로 레코드를 일괄 처리합니다. 트랜잭션이 주문되었다고 가정하면 Trident는 트랜잭션 ID를 확인하여 재시도된 배치를 정확하게 무시할 수 있습니다.

Progress-based Seep [64, 65]는 타임스탬프 비교를 사용하여 타임스탬프 순서에 따라 정확히 한 번의 출력을 제공합니다. 각 연산자는 증가하는 스칼라 타임스탬프를 생성하고 이를 레코드에 연결합니다. Seep는 운영자의 상태에 영향을 준 각 업스트림 운영자의 최신 레코드의 벡터 타임스탬프와 함께 각 운영자의 상태 및 출력을 검사합니다. 복구 시 최신 체크포인트는 새 운영자에게 로드되며, 이 운영자는 체크포인트된 출력 레코드를 재생하고 업스트림 운영자가 보낸 재생된 레코드를 처리합니다. 다운스트림 운영자는 타임스탬프를 기준으로 중복 레코드를 삭제합니다. 시스템은 시스템 시간이나 무작위 입력에 의존하지 않는 결정론적 계산을 가정합니다.

계보 기반 Timestream [129] 및 Streamscope [109]는 종속성 추적을 사용하여 정확히 한 번의 출력을 제공합니다.

정상 작동 중에 두 시스템 모두 시퀀스 번호로 레코드를 고유하게 식별하여 운영자 입력 및 출력 종속성을 추적합니다. Streamscope는 식별자와 함께 레코드를 비동기적으로 유지합니다. 두 시스템 모두 연산자 종속성을 비동기 방식으로 주기적으로 저장합니다. ~ 안에

그러나 Streamscope는 각 연산자의 종속성뿐만 아니라 상태도 개별적으로 검사합니다.

복구 시 Timestream은 상태를 재구축하는 데 필요한 모든 입력을 사용할 수 있을 때까지 업스트림 노드에 재귀적으로 연결하여 실패한 연산자의 종속성을 검색합니다.

Streamscope는 유사한 프로세스를 따르지만 실패한 운영자의 체크포인트 스냅샷에서 시작됩니다. 영구 저장소에서 찾을 수 없는 스냅샷의 각 입력 시퀀스 번호에 대해 Streamscope는 업스트림 운영자에게 연락합니다. 업스트림 운영자는 해당 시퀀스 번호가 지정된 출력 레코드를 생성할 수 있는 가장 관련성이 높은 스냅샷에서 시작하여 레코드를 다시 계산해야 할 수 있습니다. 마지막으로 두 시스템 모두 가비지 수집을 사용하여 더 이상 사용되지 않는 종속성을 삭제하지만 미묘하게 다릅니다.

예외.

Timestream은 최종 출력에서 원래 입력까지 역위상 순서로 업스트림 연산자에 필요 한 입력 레코드를 계산하고 불필요한 레코드를 삭제합니다. Streamscope는 동일한 작업을 수행하지만 종속성을 계산하는 대신 연산자 및 스트림별로 낮은 워터마크를 사용하여 오래되고 불필요한 스냅샷과 레코드를 삭제합니다.

Timestream에서 종속성을 비동기식으로 저장하면 중복된 재계산이 발생할 수 있지만 올바른 종속성 세트를 보유하는 다운스트림 연산자는 이를 삭제할 수 있습니다. Streamscope는 영구 저장소에서 중복 레코드를 찾을 수 있는 경우에만 동일한 프로세스를 적용합니다. Timestream과 Streamscope는 모두 결정론적 계산과 순서 및 값 측면의 입력을 가정합니다.

진행률 기반 및 계보 기반 솔루션은 최종 출력을 생성하는 데이터 흐름 그래프의 마지막 연산자 오류에 취약합니다. 두 솔루션 모두 중복 레코드 필터링을 위해 다운스트림 연산자에 의존하기 때문입니다.

대조적으로, 트랜잭션 기반 접근 방식은 중복 제거를 위한 다운스트림 연산자가 필요하지 않습니다. 왜냐하면 레코드의 고유 ID를 사용하여 중복 여부를 확인할 수 있기 때문입니다(표 4).

특수 싱크 연산자 Streams [56]는 파일 및 데이터베이스에서 출력을 철회하기 위한 특수 싱크를 구현합니다. 이 접근 방식을 적용하면 특정 사용 사례에 대한 출력 커밋 문제가 해결되지만 출력을 철회할 수 없다는 문제의 핵심 가정을 무시하므로 일반적으로 적용할 수 없습니다.

외부 싱크 Streams [56], Flink [37] 및 Spark [23]와 같은 일부 시스템은 상태에 대해 정확히 1회 의미 체계를 제공하고 출력 커밋 문제를 Apache Kafka와 같은 면밀성 쓰기를 지원하는 외부 싱크에 아웃소싱합니다.

특수 싱크 사업자와 외부 싱크가 제공하는 솔루션을 분류하는 한 가지 방법은 낙관적 출력 기법과 비관적 출력 기법입니다. 낙관적 출력 기술은 출력을 즉시 게시하고 철회하거나 필요한 경우 업데이트합니다. 비관적 출력 기술은 미리 쓰기 로그 형식을 사용하여 게시할 출력을 작성합니다.

on output, (2) output exactly-once, (3) precise recovery [85], and (4) strong productions [14].

Although the problem is relevant and hard, solutions in the stream-processing domain are scattered in the literature pertaining to each system in isolation. We group the various solutions into three categories: transaction-based, progress-based, and lineage-based. We now describe each of those, focusing on the assumptions they involve. Each of the three types of techniques uses a different characteristic of the input or computation, to identify whether a certain tuple has appeared previously. Transaction-based techniques use tuple identity, while progress-based techniques use order. Finally, lineage-based techniques use input–output dependencies. Finally, we provide two more categories of solutions, special sink operators and external sinks that do solve the problem in practice, but, strictly speaking, they do not meet the problem’s specification because they are either specific or external to a streaming system.

**Transaction-based** Millwheel [14] and Trident [9] rely on committing unique ids with records to eliminate duplicate retries. Millwheel assigns a unique id to each record entering the system and commits every record it produces to a highly-available storage system before sending it downstream. Downstream operators acknowledge received records. Millwheel assumes no input ordering or determinism. Trident, on the other hand, batches records into a transaction, which is assigned a unique transaction id and applies a state update to the state backend. Assuming that transactions are ordered, Trident can accurately ignore retried batches by checking the transaction id.

**Progress-based** Seep [64, 65] uses timestamp comparison to deliver exactly-once output, relying on the order of timestamps. Each operator generates increasing scalar timestamps and attaches them to records. Seep checkpoints the state and output of each operator together with the vector timestamps of the latest records from each upstream operator that affected the operator’s state. On recovery, the latest checkpoint is loaded to a new operator, which replays the checkpointed output records and processes replayed records sent by its upstream operators. Downstream operators discard duplicate records based on the timestamps. The system assumes deterministic computations that do not rely on system time or random input.

**Lineage-based** Timestream [129] and Streamscape [109] use dependency tracking to provide exactly-once output. During normal operation, both systems track operator input and output dependencies by uniquely identifying records with sequence numbers. Streamscape persists records with their identifiers asynchronously. Both systems store operator dependencies periodically in an asynchronous manner. In

Streamscape, however, each operator checkpoints individually not only its dependencies but also its state.

On recovery, Timestream retrieves the dependencies of failed operators by contacting upstream nodes recursively, until all inputs required to rebuild the state, are available. Streamscape follows a similar process, but starts from a failed operator’s checkpoint snapshot. For each input sequence number in the snapshot that is not found in persistent storage, Streamscape contacts upstream operators, which may have to recompute the record starting from their most relevant snapshot that can produce the output record given its sequence number. Finally, both systems use garbage collection to discard obsolete dependencies but in subtly different manners.

Timestream computes the input records required by upstream operators in reverse topological order from the final output to the original input and discards unneeded ones. Streamscape does the same, but instead of computing dependencies, it uses low watermarks per operator and per stream to discard older, unneeded snapshots and records. In Timestream, storing dependencies asynchronously can lead to duplicate recomputation, but downstream operators bearing the correct set of dependencies can discard them. Streamscape applies the same process only if duplicate records cannot be found in persistent storage. Both Timestream and Streamscape assume deterministic computation and input in terms of order and values.

The progress-based and lineage-based solutions are vulnerable to failures of the last operator(s) on the dataflow graph, which produces the final output, since both solutions rely on downstream operators for filtering duplicate records. In contrast, transaction-based approaches do not require a downstream operator for deduplication, since they can use the unique id of a record to check whether it is a duplicate (Table 4).

**Special sink operators** Streams [56] implements special sinks for retracting output from files and databases. The application of this approach solves the output-commit problem for specific use cases, but it is not applicable in general, since it defies the core assumption of the problem that output cannot be retracted.

**External sinks** Some systems like Streams [56], Flink [37], and Spark [23] provide exactly-once semantics on state and outsource the output-commit problem to external sinks that support idempotent writes, such as Apache Kafka.

One way to categorize the solutions provided by special sink operators and external sinks, is *optimistic* and *pessimistic output techniques*. Optimistic output techniques publish their output immediately and retract, or update it if needed. Pessimistic output techniques use a form of write-ahead log, to write the output they will publish, if everything

스트림의 진화에 관한 조사…

표 4 출력 커밋 문제를 해결하기 위해 시스템이 만드는 가정

체계	가정
밀월	처리량이 많은 외부 트랜잭션 저장소
타임스트림	결정론적 계산 및 입력
<u>스트림스코프</u>	결정론적 계산 및 입력
삼지창	결정론적 계산 및 입력, 트랜잭션 순서 지정
수록 암용 시트	결정론적 계산, 단조롭게 증가하는 논리적 시계, 타임스탬프에 따라 정렬된 레코드

출력이 영구적으로 커밋될 때까지 잘 진행됩니다 [36].

데이터베이스 세계의 다중 버전 동시성 제어와 유사한 낙관적 출력 기술에는 수정 가능하고 버전이 지정된 출력 대상이 포함되는 반면, 비관적 출력 기술에는 트랜잭션 싱크 및 유사한 도구가 포함됩니다.

상태는 파일 시스템에서 로드되고 작업에 참여하기 전에 메모리 내 상태가 재구성됩니다. 비동기식 체크포인트 외에도 새로운 체크포인트 메커니즘 [75]은 모든 다운스트림 연산자로부터 승인을 받을 때까지 출력 투플을 보존합니다. 다음으로 연산자는 출력 투플을 다듬고 체크포인트를 가져옵니다. 저자는 수동 복제가 여전히 활성 복제보다 복구 시간이 더 길지만 검사점 크기가 줄어들어 오버헤드가 90% 적다는 것을 보여줍니다.

## 5.4 고가용성

스트림 처리의 고가용성에 대한 기존 연구에서는 능동 복제 접근 방식 [30, 136], 수동 복제 접근 방식 [75, 86, 102] 또는 하이브리드 활성-수동 복제 접근 방식 [81, 143, 168]을 제안했습니다. 마지막으로 두 가지 벤치마크 평가에서는 시뮬레이션 실험을 통해 위의 접근 방식을 평가했습니다 [43, 85].

하이브리드 복제 Zwang et al. [168] 정상 작동 시 수동 모드로 작동하지만 일시적인 오류가 발생하면 일시 중단된 사전 배포된 보조 복사본을 사용하여 활성 모드로 전환하는 하이브리드 복제 접근 방식을 제안합니다. 실험 결과에 따르면 이들 접근 방식은 수동 복제에 비해 복구 시간을 66% 절약하고 활성 복제에 비해 메시지 오버헤드가 80% 적습니다.

능동 복제 Flux [136]는 계산을 복제하고 두 복제본의 진행을 조정하여 능동 복제를 구현합니다. Flux는 다른 파티션이 입력을 계속 처리하는 동안 실패한 파티션의 운영자 상태와 진행 중인 데이터를 복원합니다. 오류 발생 후 실행되는 새 기본 데이터 흐름은 새 보조 데이터 흐름이 대기 머신에 준비되면 해당 운영자의 상태를 새 보조 데이터 흐름에 복사하기 위해 정지됩니다. 대조적으로, Borealis [30]는 노드가 실패한 업스트림 노드의 라이브 복제본으로 전환하여 업스트림 노드 실패를 해결합니다. 복제본을 사용할 수 없는 경우 노드는 복구 지역을 피하기 위해 불완전한 입력에 대한 임시 출력을 생성할 수 있습니다. 이 접근 방식은 가용성을 최적화하기 위해 일관성을 희생하지만 최종 일관성을 보장합니다.

또는 Heinze et al. [81]은 장애 시 최대 대기 시간을 임계값 아래로 제한하여 시스템의 복구 오버헤드를 줄이기 위해 각 운영자에 대해 활성 복제 또는 업스트림 백업 중 복제 방식을 동적으로 선택할 것을 제안합니다. 업스트림 백업 접근 방식에서는 각 업스트림 운영자가 다운스트림 운영자가 출력 데이터를 처리할 때까지 출력 데이터를 보존합니다. 연산자가 실패하면 업스트림 연산자는 실패한 연산자의 대체 연산자가 이전 연산자의 상태를 재구성할 수 있도록 보존한 출력 데이터를 재생합니다.

수동 복제 Hwang et al. [86] 클러스터의 서버에는 체크포인트 상태의 독립적인 부분을 전달하는 백업으로 하나 이상의 서버가 있다고 제안합니다. 노드가 실패하면 체크포인트 상태의 일부를 보유하는 백업 서버는 상태가 있는 실패한 노드의 연산자를 실행하기 시작하고 체크포인트 상태에서 놓친 입력 투플을 수집하여 병렬로 복구를 시작합니다.

유사하게, Su et al. 능동적으로 복제되는 동적으로 선택된 세트를 제외하고 처리 작업을 수동적으로 복제하여 상관 오류에 대응합니다.

SGuard [102] 및 Clonos [138]는 분산 파일 시스템에 대해 상태를 비동기적으로 검사하여 대체 방식으로 계산 리소스를 절약합니다. 실패 시 실패한 운영자를 실행하기 위해 노드가 선택됩니다. 운영자의

고가용성 접근 방식의 벤치마킹 주요 작업에서 Hwang et al. [85] 다양한 유형의 쿼리 연산자에 걸쳐 활성 대기, 수동 대기, 업스트림 백업 및 기억 상실(즉, 빠른 복구를 위한 데이터 삭제)의 네 가지 복구 접근 방식의 복구 시간 및 런타임 오버헤드를 모델링하고 평가합니다. 시뮬레이션된 실험에서는 Active Standby가 리소스 활용 측면에서 높은 오버헤드를 희생하면서 거의 0에 가까운 복구 시간을 달성하는 반면, Passive Standby는 Active Standby와 비교하여 두 지표 측면에서 모두 더 나쁜 결과를 생성하는 것으로 나타났습니다. 그러나 수동적

**Table 4** Assumptions that systems make for solving the output-commit problem

System	Assumptions
Millwheel	External high-throughput transactional store
Timestream	Deterministic computation and input
Streamscape	Deterministic computation and input
Trident	Deterministic computation and input, ordering of transactions
Seep	Deterministic computation, monotonically-increasing logical clock, records ordered by timestamp

goes well, until the output is permanently committed [36]. Optimistic output techniques, which resemble multi-version concurrency control from the database world, include modifiable and versioned output destinations, while pessimistic output techniques include transactional sinks and similar tools.

## 5.4 High availability

Existing studies of high availability in stream processing proposed an active replication approach [30, 136], a passive replication approach [75, 86, 102], or a hybrid active-passive replication approach [81, 143, 168]. Finally, two benchmark evaluations assessed the approaches above, under simulated experiments [43, 85].

**Active replication** Flux [136] implements active replication by duplicating the computation and coordinating the progress of the two replicas. Flux restores operator state and in-flight data of a failed partition while the other partition continues to process input. A new primary dataflow that runs following a failure quiesces when a new secondary dataflow is ready in a standby machine, in order to copy the state of its operators to the new secondary. In contrast, Borealis [30] has nodes address upstream node failures by switching to a live replica of the failed upstream node. If a replica is not available, the node can produce tentative output for incomplete input to avoid the recovery delay. The approach sacrifices consistency to optimize availability, but guarantees eventual consistency.

**Passive replication** Hwang et al. [86] propose that a server in a cluster has one or more servers as backup where it ships independent parts of its checkpointed state. When a node fails, its backup servers that hold parts of its checkpointed state initiate recovery in parallel by starting to execute the operators of the failed node whose state they have and collecting the input tuples they have missed from the checkpointed state they possess.

SGuard [102] and Clonos [138] save computational resources in an alternative fashion, by checkpointing state asynchronously to a distributed file system. Upon failure, a node is selected to run a failed operator. The operator's

state is loaded from the file system and its in-memory state is reconstructed before it can join the job. Beyond asynchronous checkpointing, a new checkpoint mechanism [75] preserves output tuples until an acknowledgment is received from all downstream operators. Next, an operator trims its output tuples and takes a checkpoint. The authors show that passive replication still requires longer recovery time than active replication, but with 90% less overhead due to reduced checkpoint size.

**Hybrid replication** Zwang et al. [168] propose a hybrid approach to replication, which operates in passive mode under normal operation, but switches to active mode using a suspended pre-deployed secondary copy when a transient failure occurs. According to their experimental results, their approach saves 66% recovery time compared to passive replication and produces 80% less message overhead than active replication.

Alternatively, Heinze et al. [81] propose to dynamically choose the replication scheme for each operator, either active replication or upstream backup, in order to reduce the recovery overhead of the system by limiting the peak latency under failure below a threshold. The upstream backup approach maintains that each upstream operator preserves its output data until its downstream operators process them. If an operator fails, the upstream operators will replay the output data they preserve, such that the substitute operator of the failed operator can reconstruct the state of its predecessor. Similarly, Su et al. [143] counter correlated failures by passively replicating processing tasks, except for a dynamically selected set that is actively replicated.

**Benchmarking of high availability approaches** In their seminal work, Hwang et al. [85] model and evaluate the recovery time and runtime overhead of four recovery approaches, active standby, passive standby, upstream backup, and amnesia (i.e., dropping data for faster recovery), across different types of query operators. The simulated experiments suggest that active standby achieves near-zero recovery time at the expense of high overhead in terms of resource utilization, while passive standby produces worse results in terms of both metrics compared to active standby. However, passive

대기는 임의 쿼리 네트워크에 대한 유일한 옵션을 제공합니다. 업스트림 백업은 복구 시간이 길어지는 대신 런타임 오버헤드가 가장 낮습니다. 비슷한 목표를 가지고 분산 시스템 에뮬레이터인 Shrink [43]는 가동 시간 SLA 및 리소스 예약과 관련하여 다섯 가지 탄력성 전략 모델을 평가합니다. 전략은 단일 노드와 다중 노드, 활성 복제와 수동 복제, 체크포인트와 재생 등 세 가지 축에 걸쳐 다릅니다. Trill [44]을 사용하여 실제 광고 데이터에 대한 실제 쿼리를 사용한 실험에 따르면, 모든 워크로드에 적합한 단일 전략은 없지만 주기적인 체크포인트를 사용한 활성 복제가 많은 스트리밍 워크로드에서 유리한 것으로 입증되었습니다.

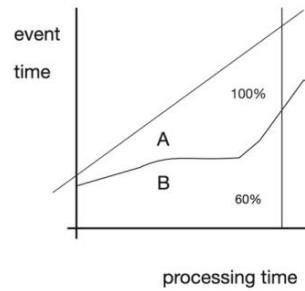


그림 5 시간에 따른 처리 시간과 이벤트 시간의 차이를 이용한 가용성 측정

## 5.5 1세대 대 2세대

초기에 스트리밍 시스템은 대량적인(최대 또는 최소 한 번) 결과에 중점을 두는 반면, 최신 시스템은 오류 발생 시 상태에 대해 정확히 한 번 처리 의미 체계를 유지합니다. 과거 시스템은 주로 상태 관리 측면으로 인해 일관성 측면에서 부족했지만 출력 커밋 문제를 해결하려고 노력했습니다.

대신, 주목을 받고 있는 최신 시스템의 일반적인 방법은 출력 중복 제거를 외부 시스템에 아웃소싱하는 것입니다. 마지막으로, 스트리밍 시스템은 오류로부터 복구하는 다운스트림 운영자가 튜플을 재생할 수 있도록 출력을 저장하는 데 사용되었지만 이제 시스템은 입력 하위 집합을 재생하기 위해 재생 가능한 입력 소스에 점점 더 의존하고 있습니다.

동시에 활성 복제를 사용하여 고가용성을 구현하는 것이 매우 일반적입니다. 이와 대조적으로 최신 시스템은 특히 클라우드 설정에 적합한 필요에 따라 추가 리소스를 할당하여 수동 복제를 활용하는 경향이 있습니다.

## 5.6 미해결 문제

스트리밍 시스템의 내결함성 및 고가용성 범위에서 세 가지 공개 문제를 강조합니다. 이는 출력 커밋 문제에 대한 새로운 솔루션, 스트리밍 처리의 가용성 정의 및 측정, 다양한 애플리케이션 요구 사항에 대한 가용성 구성과 관련됩니다.

첫째, 스트리밍 시스템이 이벤트 중심 애플리케이션 실행과 같은 새로운 방식으로 사용됨에 따라 출력 커밋 문제의 중요성이 높아질 전망입니다.

다섯 가지 유형의 솔루션을 제시했지만 이러한 솔루션은 계산 비용, 강력한 가정, 제한된 적용성 및 출력 결과의 신선도 문제로 어려움을 겪고 있습니다. 이러한 차원에서 더 나은 점수를 얻을 수 있는 새로운 종류의 솔루션이 필요합니다.

둘째, 스트리밍 처리의 고가용성에 관한 문헌은 수년에 걸쳐 스트리밍 시스템의 가용성을 크게 향상시켰습니다. 그러나 우리가 아는 한, 특히 스트리밍 처리의 가용성 의미에 대한 연구는 거의 없었습니다. 컴퓨터 시스템의 가용성에 대한 일반적인 정의

Grayet al. [73] 가용성은 단지 오류와 관련이 있습니다. 정의에 따르면 시스템은 올바른 결과로 요청에 응답할 때 사용 가능하며, 이를 서비스 성취라고 합니다. 그러나 스트리밍 처리에서는 처리가 연속적이고 잠재적으로 무제한입니다. 올바른 결과로 대응하는 것이 더욱 어려워집니다.

스트리밍의 가용성을 저하시킬 수 있는 요인에는 소프트웨어 및 하드웨어 오류, 과부하, 역압력, 처리 지연 유형(예: 체크포인트, 상태 마이그레이션, 가비지 수집, 외부 시스템 호출) 등이 있습니다.

이러한 요소들의 공통 분모는 시스템이 입력에 뒤쳐진다는 것입니다. 이러한 지연은 직렬성 속성이 충족되는 한 업데이트되지 않은 응답이 애플리케이션에서 적절하다고 인식될 수 있는 데이터베이스와 같은 다른 유형의 시스템에서는 문제가 되지 않을 수 있습니다. 그러나 스트리밍 시스템은 일반적으로 새로운 결과를 제공하기 위해 입력을 지속적으로 처리해야 합니다.

이 설문 조사는 다음과 같이 스트리밍 처리의 가용성에 대한 보다 세련된 정의에 기여합니다. 스트리밍 시스템은 가장 최근의 입력 처리에 해당하는 출력을 제공할 수 있을 때 사용 가능합니다. 이 정의는 가용성 측정 방법에 영향을 미칩니다. 적절한 방법은 시간에 따른 처리 시간과 이벤트 시간 사이의 여유 시간과 같은 진행 추적 메커니즘을 사용하는 것입니다. 이는 그림 5에 따라 입력과 관련하여 시스템의 처리 진행 상황을 정량화합니다. 플롯은 이벤트 시간과 처리 사이의 여유 시간을 나타냅니다. 시간이 지남에 따라 A를 포함하는 표면의 가용성은 100%인 반면, B를 포함하는 표면의 가용성은 60%입니다.

마지막으로 가용성은 스트리밍 시스템의 주요 비기능적 특성이며 우리가 살펴본 것처럼 추론하기가 쉽지 않습니다. 시스템이 작동하는 동안 총족할 계약으로 가용성을 지정하는 사용자 친화적인 방법을 제공하면 프로덕션 환경에서 스트리밍 시스템의 위치를 크게 향상시킬 수 있습니다. 이러한 방식으로 가용성을 구성하면 리소스 활용도, 일반 작업 중 성능 오버헤드, 복구 시간 및 일관성에 영향을 미칠 가능성이 있습니다.

standby poses the only option for arbitrary query networks. Upstream backup has the lowest runtime overhead at the expense of longer recovery time. With a similar goal, Shrink [43], a distributed systems emulator, evaluates the models of five different resiliency strategies with respect to uptime SLA and resource reservation. The strategies differ across three axes, single-node vs multi-node, active vs passive replication, and checkpoint vs replay. According to the experiments with real queries on real advertising data using Trill [44], active replication with periodic checkpoints is proved advantageous in many streaming workloads, although no single strategy is appropriate for all workloads.

## 5.5 First generation versus second generation

In the early years, streaming systems put emphasis on approximate (at-most- or at-least-once) results, while modern systems maintain exactly-once processing semantics over their state under failures. Although past systems lacked in terms of consistency, mainly due to state management aspects, they strived to solve the output-commit problem. Instead, a typical avenue for modern systems that is gaining traction is to outsource the deduplication of output to external systems. Finally, while streaming systems used to store their output to enable replaying tuples to downstream operators recovering from a failure, now, systems increasingly rely on replayable input sources for replaying input subsets.

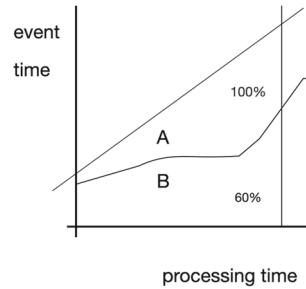
At the same time, it was very common to implement high availability using active replication. In contrast, modern systems tend to leverage passive replication, especially by allocating extra resources on demand, which is appropriate in Cloud settings.

## 5.6 Open problems

We highlight three open problems in the scope of fault tolerance and high availability in streaming systems. These regard novel solutions to the output-commit problem, defining and measuring availability in stream processing, and configuring availability for different application requirements.

First, the importance of the output-commit problem has the prospect to increase as streaming systems are used in novel ways, such as running event-driven applications. Although we presented five different types of solutions, these suffer from computational cost, strong assumptions, limited applicability, and freshness of output results. New classes of solutions are required that score better in these dimensions.

Second, the literature of high availability in stream processing has significantly enhanced the availability of streaming systems throughout the years. But, to the best of our knowledge, there has been scant research on the semantics of availability for stream processing in particular. The generic definition of availability for computer systems by



**Fig. 5** Measuring availability with the slack between processing time and event time over time

Gray et al. [73] relates availability merely to failures. According to the definition, a system is available when it responds to requests with correct results, which is termed as service accomplishment. In stream processing however, processing is continuous and potentially unbounded. Responding with correct results becomes more challenging.

The factors that may impair availability in streaming include software and hardware failures, overload, backpressure, and types of processing stall, such as checkpoints, state migration, garbage collection, and calls to external systems. The common denominator among those factors, is that the system falls behind input. This lag may not be a problem for other types of systems, such as databases where non updated responses can be perceived as adequate by applications, as long as the serializability property is satisfied. Streaming systems, though, are typically expected to continuously keep processing up with the input in order to provide fresh results.

This survey contributes a more refined definition of availability for stream processing as follows. *A streaming system is available when it can provide output corresponding to the processing of its most recent input.* This definition affects how availability is measured. An appropriate way would be via progress tracking mechanisms, such as *the slack between processing time and event time over time*, which quantifies the system's processing progress with respect to the input as per Fig. 5. The plot depicts the slack between event time and processing time over time. The surface enclosing A amounts to 100% availability, while the surface containing B equals 60% availability.

Finally, availability is a prime non-functional characteristic of a streaming system and non-trivial to reason about, as we have shown. Providing user-friendly ways to specify availability, as a contract that the system will satisfy during its operation, can significantly improve the position of streaming systems in production environments. Configuring availability in this way has the potential to impact resource utilization, performance overhead during normal operation, recovery time, and consistency.

## 6 로드 관리, 탄력성 및 재구성

외부 데이터 소스의 스트리밍 입력의 푸시 기반 특성으로 인해 스트림 프로세서는 들어오는 이벤트의 속도를 제어할 수 없습니다. 워크로드 변화 하에서 서비스 품질(QoS)을 만족시키는 것은 스트림 처리 시스템에서 오랜 연구 과제였습니다.

입력 속도가 시스템 용량을 초과할 때 성능 저하를 방지하려면 스트림 프로세서가 로드를 유지하는 조치를 취해야 합니다. 그러한 작업 중 하나는 로드 차단입니다. 즉, 스트리밍 실행 그래프의 입력 또는 중간 연산자에서 초과 튜플을 일시적으로 삭제합니다. 로드 차단은 지속 가능한 성능을 위해 결과 정확도를 상충하며 대략적인 결과를 허용할 수 있는 엄격한 대기 시간 제약이 있는 애플리케이션에 적합합니다.

짧은 대기 시간보다 결과 정확성이 더 중요한 경우 튜플 삭제는 옵션이 아닙니다. 로드 증가가 일시적인 경우 시스템은 초과 데이터를 안정적으로 버퍼링하고 입력 속도가 안정되면 나중에 처리하도록 선택할 수 있습니다. 몇몇 시스템은 생산자와 소비자가 관련된 통신 네트워크에 적용할 수 있는 기본 부하 관리 기술인 배압을 사용합니다. 그럼에도 불구하고 로드 급증 중에 사용 가능한 메모리가 부족해지는 것을 방지하기 위해 로드 인식 예약 및 속도 제어를 적용할 수 있습니다.

가변적인 입력 부하 하에서 결과 정확성을 보장하면서 QoS를 만족시키는 것을 목표로 하는 보다 최근의 접근 방식은 탄력성입니다. 탄력적 스트림 프로세서는 로드에 따라 구성을 조정하고 리소스 할당을 확장할 수 있습니다. 동적 확장 방법은 중앙 집중식 설정과 분산 설정 모두에 적용 가능합니다. 탄력성은 로드가 증가하는 경우를 해결할 뿐만 아니라 입력 로드가 감소할 때 사용되지 않는 리소스가 없도록 추가로 보장할 수 있습니다.

다음으로 로드 차단(섹션 6.1), 로드 인식 스케줄링 및 흐름 제어(섹션 6.2) 및 탄력성 기술(섹션 6.3)을 검토합니다. 이전 섹션에서와 마찬가지로 1세대 시스템과 최신 시스템 및 공개 문제에 대한 논의로 결론을 내립니다.

### 6.1 부하차단

부하 차단 [28, 144, 145, 154]은 입력 속도가 시스템 용량을 초과할 때 데이터를 폐기하는 프로세스입니다.

시스템은 쿼리 성능을 지속적으로 모니터링하고 과부하 상황이 감지되면 QoS 사양에 따라 튜플을 선택적으로 삭제합니다. 로드 차단은 일반적으로 스트림 프로세서와 통합된 독립형 구성 요소에 의해 구현됩니다. 로드 차단기는 입력 속도 또는 기타 시스템 메트릭을 지속적으로 모니터링하고 실행 중인 쿼리 계획에 대한 정보에 액세스할 수 있습니다. 주요 기능은 과부하 (부하를 차단할 시기)를 감지하고 수용을 유지하기 위해 취해야 할 조치를 결정하는 것으로 구성됩니다.

지연 시간을 확보하고 결과 품질 저하를 최소화합니다. 이러한 작업에서는 쿼리 계획의 위치, 개수, 삭제할 튜플에 대한 질문에 대답한다고 가정합니다.

잘못 트리거된 쉐딩 작업으로 인해 불필요한 결과 저하가 발생할 수 있으므로 과부하를 감지하는 것은 중요한 작업입니다. 언제 결정을 용이하게 하기 위해 로드 차단 구성 요소는 실행 중에 수집된 통계에 의존합니다. 로드 차단기가 쿼리 계획 및 실행에 대해 더 많은 지식을 갖고 있을수록 더 정확한 결정을 내릴 수 있습니다. 이러한 이유로 많은 스트림 프로세서는 튜플을 수정하지 않는 연산자(예: 필터, 결합 및 조인)와 같은 미리 정의된 연산자 집합으로 로드 차단을 제한합니다 [57, 90, 144]. 다른 연산자 제한 로드 차단 기술은 창 연산자 [28, 146]를 대상으로 하며, 더 구체적으로는 SUM 또는 COUNT 슬라이딩 창 집계 [28]를 사용하는 쿼리 계획을 대상으로 합니다. 대안적이고 운영자 독립적인 접근 방식은 피드백 제어 문제로 프레임 로드 차단을 사용하는 것입니다 [154]. 로드 차단기는 평균 튜플 지연(대기 시간)과 입력 속도 간의 관계를 설명하는 동적 모델을 사용합니다.

부하 차단기가 과부하를 감지하면 실제 부하 차단을 수행해야 합니다. 여기에는 쿼리 계획에서 튜플을 삭제할 위치와 튜플 및 수에 대한 결정이 포함됩니다. 쿼리 계획에서 특수 삭제 연산자를 가장 좋은 위치에 배치하는 것과 동등한 위치에 대한 질문입니다. 일반적으로 삭제 연산자는 쿼리 계획의 어느 위치에나 배치할 수 있지만 소스나 소스 근처에 배치되는 경우가 많습니다. 튜플을 일찍 삭제하면 나중에 작업 낭비를 피할 수 있지만 스트림 프로세서가 공유 쿼리 네트워크에서 작동하는 경우 여러 쿼리 결과에 영향을 미칠 수 있습니다.

또는 LSRM(Load Shedding Road Map)을 사용할 수도 있다 [144]. 이지도 물질화된 부하 차단 계획을 포함하고 있으며, 이로 인해 발생할 부하 차단량에 따라 정렬된 사전 계산된 테이블입니다.

어떤 튜플을 삭제할 것인지에 대한 질문은 로드 차단이 결과 품질과 관련하여 튜플의 의미론적 중요성을 고려할 때 관련됩니다. 슬라이딩 윈도우 집계 쿼리에는 쿼리 계획에 무작위 샘플링 연산자를 삽입하여 대략적인 결과를 제공하기 위해 무작위 삭제 전략이 적용되었습니다 [28]. Window-aware load shedding [146]은 개별 튜플 대신 전체 창에 차단을 적용하는 반면, 개념 기반 로드 차단 [92]은 개념 드리프트 개념을 기반으로 폐기할 튜플을 선택하는 의미론적 삭제 전략입니다.

### 6.2 스케줄링과 흐름 제어

로드 버스트가 일시적이고 일시적인 지연 증가가 결과 누락보다 선호되는 경우 역압 및 흐름 제어는 정확성을 저하시키지 않고 로드 관리를 제공할 수 있습니다. 흐름 제어 방법에는 초과 로드 버퍼링, 백로그 최소화를 목표로 운영자에게 우선순위를 부여하는 로드 인식 스케줄링, 전송 속도 조절, 생산자 제한 등이 포함됩니다. 흐름 제어 및

## 6 Load management, elasticity, and reconfiguration

Due to the push-based nature of streaming inputs from external data sources, stream processors have no control over the rate of incoming events. Satisfying Quality of Service (QoS) under workload variations has been a long-standing research challenge in stream processing systems.

To avoid performance degradation when input rates exceed system capacity, the stream processor needs to take actions that will ensure sustaining the load. One such action is *load shedding*: temporarily dropping excess tuples from inputs or intermediate operators in the streaming execution graph. Load shedding trades off result accuracy for sustainable performance and is suitable for applications with strict latency constraints that can tolerate approximate results.

When result correctness is more critical than low latency, dropping tuples is not an option. If the load increase is transient, the system can instead choose to reliably buffer excess data and process it later, once input rates stabilize. Several systems employ *back-pressure*, a fundamental load management technique applicable to communication networks that involve producers and consumers. Nevertheless, to avoid running out of available memory during load spikes, *load-aware scheduling* and rate control can be applied.

A more recent approach that aims at satisfying QoS while guaranteeing result correctness under variable input load is *elasticity*. Elastic stream processors are capable of adjusting their configuration and scaling their resource allocation in response to load. Dynamic scaling methods are applicable to both centralized and distributed settings. Elasticity not only addresses the case of increased load, but can additionally ensure no resources are left unused when the input load decreases.

Next, we review load shedding (Sect. 6.1), load-aware scheduling and flow control (Sect. 6.2), and elasticity techniques (Sect. 6.3). As in previous sections, we conclude with a discussion of first generation vs. modern systems and open problems.

### 6.1 Load shedding

Load shedding [28, 144, 145, 154] is the process of discarding data when input rates increase beyond system capacity. The system continuously monitors query performance and if an overload situation is detected, it selectively drops tuples according to a QoS specification. Load shedding is commonly implemented by a standalone component integrated with the stream processor. The load shedder continuously monitors input rates or other system metrics and can access information about the running query plan. Its main functionality consists of detecting overload (*when* to shed load) and deciding what actions to take in order to maintain accept-

able latency and minimize result quality degradation. These actions presume answering the questions of *where* (in the query plan), *how many*, and *which* tuples to drop.

Detecting overload is a crucial task, as an incorrectly triggered shedding action can cause unnecessary result degradation. To facilitate the decision of *when*, load shedding components rely on statistics gathered during execution. The more knowledge a load shedder has about the query plan and its execution, the more accurate decisions it can make. For this reason, many stream processors restrict load shedding to a predefined set of operators, such as those that do not modify tuples, i.e. filter, union, and join [57, 90, 144]. Other operator-restricted load shedding techniques target window operators [28, 146], or even more specifically, query plans with SUM or COUNT sliding window aggregates [28]. An alternative, operator-independent approach is to frame load shedding as a feedback control problem [154]. The load shedder relies on a dynamic model that describes the relationship between average tuple delay (latency) and input rate.

Once the load shedder has detected overload, it needs to perform the actual load shedding. This includes the decision of where in the query plan to drop tuples from, as well as which tuples and how many. The question of where is equivalent to placing special *drop operators* in the best positions in the query plan. In general, drop operators can be placed at any location in the query plan, however, they are often placed at or near the sources. Dropping tuples early avoids wasting work later, but it might affect results of multiple queries if the stream processor operates on a shared query network. Alternatively, a load shedding road map (LSRM) can be used [144]. This map is a pre-computed table that contains materialized load-shedding plans, ordered by the amount of load shedding they will cause.

The question of which tuples to drop is relevant when load shedding takes into account the *semantic* importance of tuples with respect to results quality. A *random* dropping strategy has been applied to sliding window aggregate queries to provide approximate results by inserting random sampling operators in the query plan [28]. *Window-aware* load shedding [146] applies shedding to entire windows instead of individual tuples, while *concept-driven* load shedding [92] is a semantic dropping strategy that selects tuples to discard based on the notion of concept drift.

### 6.2 Scheduling and flow control

When load bursts are transient and a temporary increase in latency is preferred to missing results, back-pressure and flow control can provide load management without sacrificing accuracy. Flow control methods include buffering excess load, load-aware scheduling that prioritizes operators with the objective to minimize the backlog, regulating the transmission rate, and throttling the producer. Flow control and

배압 기술은 입력 투플의 의미론적 중요성과 같은 애플리케이션 수준 품질 요구 사항을 고려하지 않습니다. 주요 요구 사항은 소스 또는 중간 운영자의 버퍼 공간 가용성과 누적된 로드가 시스템 용량 제한 내에 있어야 최종적으로 데이터 백로그를 처리할 수 있다는 것입니다.

로드 인식 스케줄링은 연산자 실행 순서를 선택하고 리소스 할당을 조정하여 과부하 문제를 해결합니다. 예를 들어, 필터 및 조인 실행 순서를 동적으로 선택하여 백로그를 줄일 수 있습니다 [25, 29]. 대안으로 적응형 스케줄링 [26, 40]은 정적 쿼리 계획에 따라 리소스 할당을 수정합니다. 로드 인식 스케줄링 전략의 목적은 시스템의 전체 입력 대기열 크기를 최소화하는 운영자 실행 순서를 선택하는 것입니다. 스케줄러는 운영자 선택성 및 처리 비용에 대한 지식에 의존합니다.

이러한 통계는 사전에 알려진 것으로 가정되거나 런타임 중에 주기적으로 수집되어야 합니다. 운영자에게는 중간 결과를 최소화할 수 있는 잠재력과 결과적으로 대기열 크기를 반영하는 우선순위가 할당됩니다. 온라인 셜플 그룹화 [132]는 균일하지 않은 투플 실행 시간으로 인한 불균형을 줄이는 것을 목표로 하는 적응형 투플별 스케줄링 기술입니다. 이는 스케치를 사용하여 투플 실행 시간을 지속적으로 모니터링하고 그리디 스케줄링 알고리즘을 사용하여 온라인 방식으로 병렬 작업에 투플을 할당합니다.

역압 및 흐름 제어 다수의 연산자가 있는 스트리밍 실행 그래프와 같은 소비자와 생산자의 네트워크에서 역압은 가장 느린 소비자의 처리 속도에 맞춰 모든 연산자를 느리게 만드는 효과가 있습니다. 병목 현상 연산자가 데이터 흐름 그래프 아래에 있는 경우 역압이 업스트림 연산자로 전파되어 결국 데이터 스트림 소스에 도달합니다. 데이터 손실을 방지하려면 Apache Kafka와 같은 영구 입력 메시지 대기열과 적절한 저장 공간이 필요합니다.

버퍼 기반 역압력은 버퍼 가용성을 통해 데이터 흐름을 암시적으로 제어합니다. 고정된 양의 버퍼 공간을 고려하면 병목 현상 연산자는 버퍼가 데이터 흐름 경로를 따라 점차적으로 채워지게 만듭니다. 그림 6a는 생산자와 소비자가 동일한 시스템에서 실행되고 버퍼 풀을 공유할 때 버퍼 기반 흐름 제어를 보여줍니다. 생산자는 결과를 생성하면 이를 출력 버퍼로 직렬화합니다. 생산자와 소비자가 동일한 시스템에서 실행되고 소비자가 느린 경우 생산자는 사용 가능한 출력 버퍼가 없을 때 출력 버퍼를 검색하려고 시도할 수 있습니다. 따라서 생산자의 처리 속도는 소비자가 버퍼를 공유 버퍼 풀로 다시 재활용하는 속도에 따라 느려집니다. 생산자와 소비자가 서로 다른 시스템에 배포되고 TCP를 통해 통신하는 경우가 그림 6b에 나와 있습니다. 소비자 측에서 사용 가능한 버퍼가 없으면 TCP 창 메커니즘은 발신자에게 다음을 알립니다.

데이터 전송을 중단합니다. 생산자는 임계값을 사용하여 전송 중인 데이터의 양을 제어할 수 있으며, 새 데이터를 연결할 수 없는 경우 속도가 느려집니다.

CFC( Credit-Based Flow Control ) [101]는 ATM 네트워크 스위치에 사용되는 링크별 가상 채널별 혼잡 제어 기술입니다. 간단히 말해서 CFC는 신용 시스템을 사용하여 수신자에서 발신자에게 버퍼 공간의 가용성을 알립니다. 이 고전적인 네트워킹 기술은 현대의 고도로 병렬화된 스트림 프로세서의 로드 관리에 매우 유용한 것으로 밝혀졌으며 Apache Flink [1]에서 구현되었습니다. 그림 7은 가상 데이터 흐름에 대해 이 체계가 어떻게 작동하는지 보여줍니다. 병렬 작업은 TCP 연결을 통해 다중화된 가상 채널을 통해 연결됩니다. 각 작업은 신용 메시지를 통해 보낸 사람에게 버퍼 가용성을 알립니다. 이런 방식으로 발신자는 수신자가 데이터 메시지를 처리하는 데 필요한 용량을 갖추고 있는지 항상 알 수 있습니다. 수신기의 크레딧이 0(또는 지정된 임계값)으로 떨어지면 배압이 가상 채널에 나타납니다. 이 채널별 흐름 제어 메커니즘의 중요한 장점은 배압이 통신 작업 쌍에만 적용되고 동일한 TCP 연결을 공유하는 다른 작업을 방해하지 않는다는 것입니다. 이 측면은 단일 오버로드된 작업이 다른 모든 다운스트림 운영자 인스턴스로의 데이터 흐름을 차단할 수 있는 데이터 왜곡이 있는 경우 중요합니다.

단점은 추가 신용 발표 메시지로 인해 종단 간 대기 시간이 늘어날 수 있다는 점입니다.

### 6.3 탄력성

로드 차단 및 역압 접근 방식은 정적으로 프로비저닝된 스트림 프로세서 또는 애플리케이션의 워크로드 변화를 처리하도록 설계되었습니다. 클라우드 환경이나 클러스터에 배포된 스트림 프로세서는 동적 리소스 풀을 활용할 수 있습니다. 동적 확장 또는 탄력성은 워크로드 변화를 효율적으로 처리하기 위해 실행 중인 계산에 사용 가능한 리소스를 변경하는 스트림 프로세서의 기능입니다. 탄력적인 스트리밍 시스템을 구축하려면 정책과 메커니즘이 필요합니다. 정책 구성 요소는 성능 지표를 수집하고 확장 시기와 규모를 결정하는 제어 알고리즘을 구현합니다. 메커니즘은 구성 변경에 영향을 미칩니다. 리소스 할당, 작업 재할당, 상태 마이그레이션을 처리하는 동시에 결과의 정확성을 보장합니다. 표 5에는 탄력적 스트리밍 시스템의 동적 확장 기능과 특성이 요약되어 있습니다.

#### 6.3.1 탄력성 정책

조정 정책에는 두 가지 개별 결정이 포함됩니다. 먼저, 건강하지 않거나 비효율적인 계산의 증상을 감지하고 크기 조정이 필요한지 결정해야 합니다.

증상 감지는 잘 알려진 문제이며 기존 모니터링 도구를 사용하여 해결할 수 있습니다. 두 번째,

back-pressure techniques do not consider application-level quality requirements, such as the semantic importance of input tuples. Their main requirement is availability of buffer space at the sources or intermediate operators and that any accumulated load is within the system capacity limits, so that it will be eventually possible to process the data backlog.

**Load-aware scheduling** tackles the overload problem by selecting the *order* of operator execution and by adapting the *resource allocation*. For instance, backlog can be reduced by dynamically selecting the order of executing filters and joins [25, 29]. Alternatively, adaptive scheduling [26, 40] modifies the allocation of resources given a static query plan. The objective of load-aware scheduling strategies is to select an operator execution order that minimizes the total size of input queues in the system. The scheduler relies on knowledge about operator selectivities and processing costs. These statistics are either assumed to be known in advance, or need to be collected periodically during runtime. Operators are assigned priorities that reflect their potential to minimize intermediate results, and, consequently, the size of queues. Online shuffle grouping [132] is an adaptive per-tuple scheduling technique that aims to reduce the imbalance caused by non-uniform tuple execution times. It relies on sketches to continuously monitor tuple execution times and uses a greedy scheduling algorithm to assign tuples to parallel tasks in an online fashion.

**Back-pressure and flow control** In a network of consumers and producers such as a streaming execution graph with multiple operators, back-pressure has the effect that all operators slow down to match the processing speed of the slowest consumer. If the bottleneck operator is far down the dataflow graph, back-pressure propagates to upstream operators, eventually reaching the data stream sources. To ensure no data loss, a persistent input message queue, such as Apache Kafka, and adequate storage space are required.

*Buffer-based* back-pressure implicitly controls the flow of data via buffer availability. Considering a fixed amount of buffer space, a bottleneck operator will cause buffers to gradually fill up along its dataflow path. Figure 6a demonstrates buffer-based flow control when the producer and the consumer run on the same machine and share a buffer pool. When a producer generates a result, it serializes it into an output buffer. If the producer and consumer run on the same machine and the consumer is slow, the producer might attempt to retrieve an output buffer when none is available. The producer's processing rate will, thus, slow down according to the rate the consumer is recycling buffers back into the shared buffer pool. The case when the producer and consumer are deployed on different machines and communicate via TCP is shown in Fig. 6b. If no buffer is available on the consumer side, the TCP window mechanism will inform the sender to

halt data transmission. The producer can use a threshold to control how much data is in-flight and it is slowed down if it cannot put new data on the wire.

*Credit-based flow control* (CFC) [101] is a link-by-link, per virtual channel congestion control technique used in ATM network switches. In a nutshell, CFC uses a credit system to signal the availability of buffer space from receivers to senders. This classic networking technique turns out to be very useful for load management in modern, highly-parallel stream processors and is implemented in Apache Flink [1]. Figure 7 shows how the scheme works for a hypothetical dataflow. Parallel tasks are connected via virtual channels multiplexed over TCP connections. Each task informs its senders of its buffer availability via credit messages. This way, senders always know whether receivers have the required capacity to handle data messages. When the credit of a receiver drops to zero (or a specified threshold), back-pressure appears on its virtual channel. An important advantage of this per-channel flow control mechanism is that back-pressure is inflicted on pairs of communicating tasks only and does not interfere with other tasks sharing the same TCP connection. This aspect is crucial in the presence of data skew, where a single overloaded task could otherwise block the flow of data to all other downstream operator instances. On the downside, the additional credit announcement messages might increase end-to-end latency.

## 6.3 Elasticity

The approaches of load shedding and back-pressure are designed to handle workload variations in a *statically provisioned* stream processor or application. Stream processors deployed on cloud environments or clusters can make use of a dynamic pool of resources. *Dynamic scaling* or *elasticity* is the ability of a stream processor to vary the resources available to a running computation in order to handle workload variations efficiently. Building an elastic streaming system requires a *policy* and a *mechanism*. The policy component implements a control algorithm that collects performance metrics and decides when and how much to scale. The mechanism effects the configuration change. It handles resource allocation, work re-assignment, and state migration, while guaranteeing result correctness. Table 5 summarizes the dynamic scaling capabilities and characteristics of elastic streaming systems.

### 6.3.1 Elasticity policies

A *scaling policy* involves two separate decisions. First, it needs to detect the symptoms of an unhealthy or inefficient computation and decide whether scaling is necessary. Symptom detection is a well-understood problem and can be addressed using conventional monitoring tools. Second,

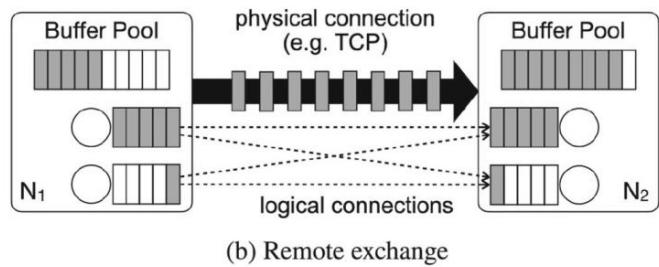
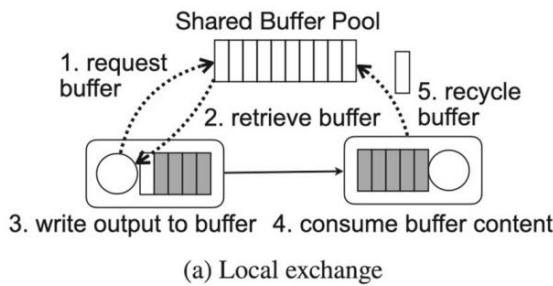


그림 6 버퍼 기반 흐름 제어. 원은 운영자 작업을 나타내고 가장자리는 데이터 종속성을 나타냅니다. 지역교류의 경우 업무는 동일한 물리적 노드의 프로세스. 원격 교환의 경우 생산자와 소비자 작업은 별도의 물리적 노드에서 프로세스됩니다.  
(N1 및 N2로 표시됨)

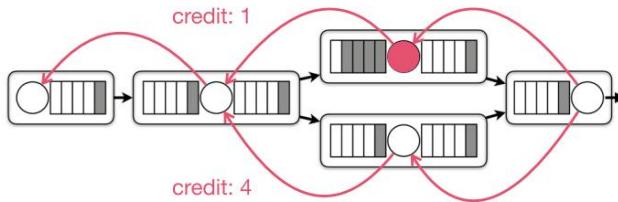


그림 7 데이터 흐름 그래프의 크레딧 기반 흐름 제어. 수신자는 정기적으로 자신의 신용 업스트림을 발표합니다(회색 및 흰색 사각형은 전체 버퍼와 여유 버퍼 각각)(온라인 커리 그림)

정책은 나타난 증상의 원인(예: 병목 현상 운영자)을 식별하고 규모 조정을 제안해야 합니다.

행동. 이는 성능 분석과 예측이 필요한 어려운 작업입니다. 하는 것이 일반적인 관행입니다.

확장 결정의 부담을 애플리케이션 사용자에게 전가

상충되는 인센티브에 직면해야 하는 사람. 그들은 다음 중 하나를 계획할 수 있습니다.

예상되는 가장 높은 작업 부하에 대해 높은 비용이 발생할 수 있음

비용을 절감하거나 보수적이고 위험을 낮추는 방식을 선택할 수 있습니다.

성능. 자동 확장은 확장 결정을 의미합니다.

응답으로 스트리밍 시스템에 의해 투명하게 처리됨

로드합니다. 자동 기능을 제공하는 상업용 스트리밍 시스템

확장에는 Google Cloud Dataflow [95], Heron [100],

및 IBM System S [71], DS2 [89], Seep [64] 및

StreamCloud [76]는 이러한 기능을 갖춘 최근 연구 프로토타입입니다.

지원하다.

표 5에서는 정책을 휴리스틱과 예측으로 분류합니다. 경험적 정책은 경험적으로 사전 정의된

규칙을 적용하고 종종 임계값이나 관찰된 조건에 의해 트리거되는 반면 예측 정책은 안내에 따라 확장 결정을 내립니다.

분석적 성능 모델을 통해

경험적 정책 컨트롤러는 CPU 사용률, 관찰된 처리량, 대기열 등 대략적인 측정 항목을 수집합니다.

최적이 아닌 확장을 감지하기 위한 크기 및 메모리 활용도입니다. CPU 및 메모리 활용도 지표가 부적절할 수 있음

클라우드 환경에 배포된 스트리밍 애플리케이션용

다중 테넌시 및 성능 간섭으로 인해 [131].

StreamCloud [76] 및 Seep [64]는 사용자 시간과 시스템 시간을 분리하여 문제를 완화하려고 시도하지만 선점됩니다.

이러한 측정항목이 오해를 불러일으킬 수 있습니다. 예를 들어 높은 CPU

동일한 물리적 시스템에서 실행되는 작업으로 인해 발생하는 사용량 데이터 흐름 운영자가 잘못된 확장(false)을 트리거할 수 있기 때문입니다. 긍정) 또는 올바른 축소(거짓 부정)를 방지합니다.

Google Cloud Dataflow [95]는 CPU 사용률에 의존합니다.

축소 결정만 가능하지만 여전히 거짓 부정이 발생합니다.

Dhalion [66] 및 IBM Streams [71] 도 혼잡 및

병목 현상을 식별하는 배압 신호. 이러한 측정항목

병목 현상을 식별하는 데는 도움이 되지만 감지할 수는 없습니다.

리소스 과잉 프로비저닝.

예측 정책 컨트롤러는 스트리밍 시스템의 분석 성능 모델을 구축하고

수학 함수의 집합으로 스케일링 문제를 해결합니다. 예측

접근 방식에는 대기열 이론 [67, 111, 151], 제어가 포함됩니다.

이론 [18, 93, 115] 및 계측 중심 선형 성능 모델 [89]. 폐쇄형 분석 덕분에

공식화, 예측 정책을 통해 한 단계로 여러 운영자의 결정을 내릴 수 있습니다.

### 6.3.2 탄력성 메커니즘

탄력성 메커니즘은 다음을 실현하는 데 관심이 있습니다.

정책에 명시된 조치. 축적된 상태의 정확성과 낮은 지연 시간 재분배를 보장해야 합니다.

재구성을 수행할 때. 정확성을 보장하기 위해,

많은 스트리밍 시스템은 재구성 기능을 제공하기 위해 내결함성 메커니즘에 의존합니다. 추가할 때

실행 중인 계산에 새로운 작업자를 추가하려면 메커니즘이 필요합니다.

그들에게 작업을 재할당할 뿐만 아니라 이제 새로운 작업자가 담당하게 될 필요 한 모든 상태를 마이그레이션합니다.

탄력성 메커니즘은 재구성을 완료해야 합니다.

최대한 빨리 수행하는 동시에 성능 저하를 최소화합니다. 상태에 대한 주요 방법 을 검토합니다.

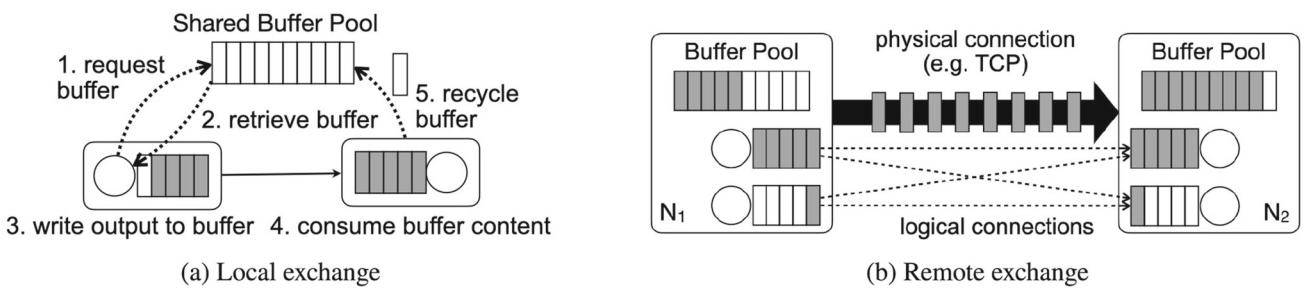
그 다음은 재배포, 재구성, 상태 이전입니다. 우리

재구성과 같이 임베디드 상태가 있는 시스템에 중점을 둡니다.

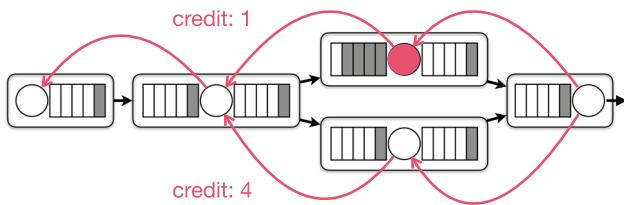
상태가 외부일 때 메커니즘은 상당히 단순화됩니다.

상태 재분배 상태 재분배는 키를 보존해야 합니다.

의미론적으로 특정 키와 모든 키에 대한 기준 상태



**Fig. 6** Buffer-based flow control. Circles denote operator tasks and edges denote data dependencies. In the case of local exchange, tasks are processes on the same physical node. In the case of remote exchange, the producer and consumer tasks are processes on separate physical nodes (shown as  $N_1$  and  $N_2$ ).



**Fig. 7** Credit-based flow control in a dataflow graph. Receivers regularly announce their credit upstream (gray and white squares indicate full and free buffers, respectively) (color figure online)

the policy needs to identify the causes of exhibited symptoms (e.g. a bottleneck operator) and propose a scaling action. This is a challenging task which requires performance analysis and prediction. It is common practice to place the burden of scaling decisions on application users who have to face conflicting incentives. They can either plan for the highest expected workload, possibly incurring high cost, or they can choose to be conservative and risk degraded performance. Automatic scaling refers to scaling decisions transparently handled by the streaming system in response to load. Commercial streaming systems that offer automatic scaling include Google Cloud Dataflow [95], Heron [100], and IBM System S [71], while DS2 [89], Seep [64] and StreamCloud [76] are recent research prototypes with such support.

In Table 5, we categorize policies into *heuristic* and *predictive*. Heuristic policies rely on empirically predefined rules and are often triggered by thresholds or observed conditions while predictive policies make scaling decisions guided by analytical performance models.

Heuristic policy controllers gather coarse-grained metrics, such as CPU utilization, observed throughput, queue sizes, and memory utilization, to detect suboptimal scaling. CPU and memory utilization can be inadequate metrics for streaming applications deployed in cloud environments due to multi-tenancy and performance interference [131]. StreamCloud [76] and Seep [64] try to mitigate the problem by separating user time and system time, but preemption can make these metrics misleading. For example, high CPU

usage caused by a task running on the same physical machine as a dataflow operator can trigger incorrect scale-ups (false positives) or prevent correct scale-downs (false negatives). Google Cloud Dataflow [95] relies on CPU utilization for scale-down decisions only but still suffers false negatives. Dhalion [66] and IBM Streams [71] also use congestion and back-pressure signals to identify bottlenecks. These metrics are helpful for identifying bottlenecks but they cannot detect resource over-provisioning.

Predictive policy controllers build an analytical performance model of the streaming system and formulate the scaling problem as a set of mathematical functions. Predictive approaches include queuing theory [67, 111, 151], control theory [18, 93, 115], and instrumentation-driven linear performance models [89]. Thanks to their closed-form analytical formulation, predictive policies are capable of making multi-operator decisions in one step.

### 6.3.2 Elasticity mechanisms

Elasticity mechanisms are concerned with realizing the actions indicated by the policy. They need to ensure correctness and low-latency redistribution of accumulated state when effecting a reconfiguration. To ensure correctness, many streaming systems rely on the fault-tolerance mechanism to provide reconfiguration capabilities. When adding new workers to a running computation, the mechanism needs not only re-assign work to them but also migrate any necessary state that these new workers will now be in charge of. Elasticity mechanisms need to complete a reconfiguration as quickly as possible and at the same time minimize performance disruption. We review the main methods for state redistribution, reconfiguration, and state transfer next. We focus on systems with embedded state, as reconfiguration mechanisms are significantly simplified when state is external.

**State redistribution** State redistribution must preserve key semantics, so that existing state for a particular key and all

표 5 스트리밍 시스템의 탄력성 정책 및 메커니즘

체계	정책	목적	재구성	상태 마이그레이션
휴리스틱 예측 대기 시간 처리량 중지 및 재시작 부분 일시 정지 Live At-Once 프로그래시브				
보레알리스 [11]				
스트리밍클라우드 [76]				
스며들다 [64]				
IBM 스트리밍 [71]				
복어 [79, 80]				
네델레 [111]				
DRS [67]				
MPC [115]				
혜성구름 [151]				
크로노스트림 [160]				
에이스 [18]				
스텔라 [161]				
구글 데이터플로우 [95]				
달리온 [66]				
DS2 [89]				
스파크 스트리밍 [23, 163]				
메가폰 [84]				
터빈 [117]				
코뿔소 [119]				

이 키를 사용하는 항후 이벤트는 동일한 작업자로 라우팅됩니다.

이를 위해 대부분의 시스템은 해싱 방법을 사용합니다. 균일한 해싱은 병렬 작업 전체에 키를 균등하게 분배합니다.

계산 속도가 빠르고 라우팅 상태가 필요하지 않지만

높은 마이그레이션 비용이 발생합니다. 새 노드가 추가되면 상태

기존 작업자와 신규 작업자 간에 섞입니다. 이는 또한

랜덤 I/O 및 높은 네트워크 통신. 따라서,

적응형 애플리케이션에는 특히 적합하지 않습니다. 일관성 있는 해싱과 변형이 선호되는 경우가 더 많습니다. 노동자

키는 다중 무작위 해시 함수를 사용하여 링의 여러 지점에 매핑됩니다. 일관된 해싱은 다음을 보장합니다.

상태는 이전에 존재했던 워커 간에 이동되지 않고

마이그레이션 후, 새로운 직원이 합류하면

기존의 여러 데이터 항목을 담당합니다.

노드. 작업자가 떠날 때 해당 키 공간이 배포됩니다.

기존 근로자보다 Apache Flink [37]는 변형을 사용합니다.

상태가 키로 구성되는 일관된 해싱

그룹은 병렬 작업에 범위로 매핑됩니다. ~에

재구성, 읽기는 각 키 그룹 내에서 순차적입니다.

그리고 종종 여러 주요 그룹에 걸쳐 발생합니다. 키 그룹-작업 할당의 메타데이터는 작으며 다음 작업에 충분합니다.

키 그룹 범위 경계를 저장합니다. 주요 그룹 수

키가 지정된 병렬 작업의 최대 수를 제한합니다.

상태를 확장할 수 있습니다.

해싱 기술은 구현 및 수행이 간단합니다.

라우팅 상태를 저장할 필요는 없지만

왜곡된 키 분포에서도 잘 수행됩니다. 하이브리드 파

tiitioning [70]은 일관된 해싱과 명사적인 해싱을 결합합니다.

다음을 제공하는 컴팩트 해시 함수를 생성하는 매핑

불균형이 있는 경우 로드 밸런싱. 주요 아이디어는 추적하는 것입니다.

분할 키 값의 빈도를 조정하고 일반 키와 인기 키를 다르게 처리합니다. 메커니즘은 다음을 사용합니다.

손실 계산 알고리즘 [114]은 슬라이딩 윈도우 설정에서 강력한 타자를 추정하기 위한 것입니다. 정확한 카운트를 유지하는 것은

큰 키 도메인에는 실용적이지 않습니다. DKG [133] 도 비슷하다.

널리 사용되는 키를 명사적으로 매핑하는 키 그룹화 메커니즘

덜 인기 있는 키 그룹과 함께 하위 스트림으로  
로드 밸런싱을 달성합니다.

재구성 전략 재파티셔닝과 관계없이

탄력성 정책이 다음을 결정하는 경우 사용되는 전략

애플리케이션의 리소스를 변경하면 메커니즘은  
동일한 작업자 간에 일정량의 상태를 전송하려면

또는 다른 물리적 기계.

중지 및 재시작 전략은 계산을 중단하고

모든 운영자의 상태 스냅샷을 만든 다음 새 구성으로 애플리케이션을 다시 시작합니다. 이런 메커니즘이에도

구현이 간단하고 정확성이 거의 보장되지 않습니다.

단 하나라도 전체 파일프라인을 불필요하게 자연사킵니다.

소수의 연산자를 재조정해야 합니다. 표 5에 나타난 바와 같이, 이는  
전략은 현대 시스템에서 일반적입니다.

FLUX [137]에 의해 도입된 부분적인 일시정지 및 재시작,

영향을 받은 대상만 차단하는 덜 파괴적인 전략입니다.

일시적으로 데이터 흐름 하위 그래프. 영향을 받은 하위 그래프 구성

**Table 5** Elasticity policies and mechanisms in streaming systems

System	Policy		Objective		Reconfiguration			State migration	
	Heuristic	Predictive	Latency	Throughput	Stop-and-restart	Partial pause	Live	At-Once	Progressive
Borealis [11]	✓		✓	✓	n/a			n/a	
StreamCloud [76]	✓			✓		✓		✓	
Seep [64]	✓		✓	✓		✓		✓	
IBM Streams [71]	✓			✓		✓		✓	
FUGU [79, 80]	✓			✓		✓		✓	
Nephele [111]		✓	✓						
DRS [67]		✓	✓						
MPC [115]		✓	✓			✓		✓	
CometCloud [151]		✓	✓				✓	n/a	
Chronostream [160]	n/a		n/a				✓	✓	
ACES [18]		✓	✓	✓	n/a			n/a	
Stella [161]	✓			✓					
Google Dataflow [95]	✓		✓	✓					
Dhalion [66]	✓			✓	✓			✓	
DS2 [89]		✓		✓	✓	✓		✓	
Spark Streaming [23, 163]	✓			✓	✓			✓	
Megaphone [84]						✓			✓
Turbine [117]	✓			✓	✓			✓	
Rhino [119]	n/a		n/a			✓		✓	

future events with this key are routed to the same worker. For that purpose, most systems use hashing methods. *Uniform hashing* evenly distributes keys across parallel tasks. It is fast to compute and requires no routing state but might incur high migration cost. When a new node is added, state is shuffled across existing and new workers. It also causes random I/O and high network communication. Thus, it is not particularly suitable for adaptive applications. *Consistent hashing* and variations are more often preferred. Workers and keys are mapped to multiple points on a ring using multiple random hash functions. Consistent hashing ensures that state is not moved across workers that are present before and after the migration. When a new worker joins, it becomes responsible for data items from multiple of the existing nodes. When a worker leaves, its key space is distributed over existing workers. Apache Flink [37] uses a variation of consistent hashing in which state is organized into *key groups* and those are mapped to parallel tasks as ranges. On reconfiguration, reads are sequential within each key group, and often across multiple key groups. The metadata of key-group-to-task assignments are small and it is sufficient to store key-group range boundaries. The number of key groups limits the maximum number of parallel tasks to which keyed state can be scaled.

Hashing techniques are simple to implement and do not require storing any routing state, however, they do not perform well under skewed key distributions. *Hybrid par-*

*titioning* [70] combines consistent hashing and an explicit mapping to generate a compact hash function that provides load balance in the presence of skew. The main idea is to track the frequencies of the partitioning key values and treat normal keys and popular keys differently. The mechanism uses the lossy counting algorithm [114] in a sliding window setting to estimate heavy hitters, as keeping exact counts would be impractical for large key domains. DKG [133] is a similar key-grouping mechanism that explicitly maps popular keys to sub-streams together with groups of less popular keys to achieve load balance.

**Reconfiguration strategy** Regardless of the re-partitioning strategy used, if the elasticity policy makes a decision to change an application’s resources, the mechanism will have to transfer some amount of state across workers on the same or different physical machines.

The *stop-and-restart* strategy halts the computation, takes a state snapshot of all operators, and then restarts the application with the new configuration. Even though this mechanism is simple to implement and it trivially guarantees correctness, it unnecessary stalls the entire pipeline even if only one or few operators need to be rescaled. As shown in Table 5, this strategy is common in modern systems.

*Partial pause and restart*, introduced by FLUX [137], is a less disruptive strategy that only blocks the affected dataflow subgraph temporarily. The affected subgraph con-

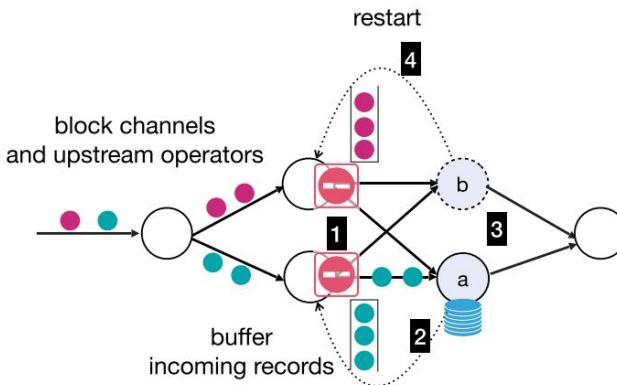


그림 8 부분 일시 정지 및 재시작 프로토콜의 예. 상태를 연산자 a에서 b로 이동하기 위해 메커니즘은 다음 단계를 실행합니다. (1) a의 업스트림 연산자를 일시 중지하고, (2) a에서 상태를 추출하고, (3) 상태를 b로 로드하고, (4) b에서 다시 시작 신호를 보냅니다. 업스트림 운영자에게

확장할 연산자와 업스트림 채널 및 업스트림 연산자를 포함합니다. 그림 8은 프로토콜의 예를 보여줍니다. 상태를 연산자 a에서 연산자 b로 마이그레이션하기 위해 메커니즘은 다음 단계를 실행합니다. (1) 먼저 a의 업스트림 연산자를 일시 중지하고 튜플을 a로 푸시하는 것을 중지합니다. 일시 중지된 연산자는 로컬 버퍼에서 입력 튜플 버퍼링을 시작합니다. 연산자 a는 버퍼가 빌 때까지 버퍼의 튜플을 계속 처리합니다. (2) a의 버퍼가 비면 상태를 추출하여 연산자 b로 보냅니다. (3) 연산자 b는 상태를 로드하고 (4) 업스트림 연산자에게 재시작 신호를 보냅니다. 업스트림 운영자가 신호를 받으면 튜플 처리를 다시 시작할 수 있습니다.

ChronoStream [160] 및 CometCloud [151]과 같은 시스템은 사전 예방적 복제 전략을 활용하여 거의 실시간 방식으로 재구성을 수행합니다. 핵심 아이디어는 여러 노드에서 상태 백업 복사본을 유지하는 것입니다. 이를 위해 상태는 더 작은 파티션으로 구성되며 각 파티션은 독립적으로 전송될 수 있습니다. 노드에는 기본 상태 조각 세트와 보조 상태 조각 세트가 있습니다. 그림 9는 ChronoStream 프로토콜의 예를 보여줍니다.

상태 이전 한 작업자에서 다른 작업자로 상태를 이전할 때 내려야 할 또 다른 중요한 결정은 상태를 한꺼번에 이동 할지 아니면 점진적 으로 이동할지 여부입니다. 많은 양의 상태를 전송해야 하는 경우 한 번의 작업으로 이동하면 재구성 중에 대기 시간이 길어질 수 있습니다.

또는 점진적 마이그레이션 [84]은 상태 전송을 처리와 함께 인터리빙하여 상태를 더 작은 조각으로 이동하고 대기 시간 스파이크를 평탄화합니다. 단점은 점진적인 상태 마이그레이션으로 인해 마이그레이션 기간이 길어질 수 있다는 것입니다.

#### 6.4 1세대 대 2세대

초기 접근 방식과 현대 접근 방식을 비교하여 다음과 같은 관찰을 합니다. 로드 차단은 초기 스트림 프로세서에서 인기가 있었지만 현대 시스템은 로드 차단을 선호하지 않습니다.

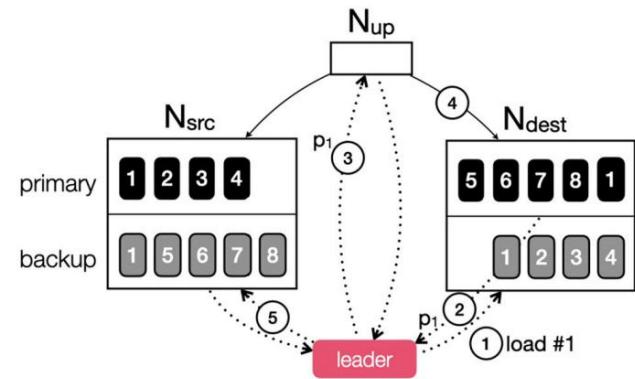


그림 9 사전 복제 프로토콜의 예. Nsrc에서 Ndest로 슬라이스 #1을 이동하기 위해 메커니즘은 다음 단계를 실행합니다. (1) 리더는 Ndest에게 슬라이스 #1을 로드하도록 지시합니다. (2) Ndest는 슬라이스 #1을 로드하고 리더에게 ack를 보냅니다. (3) 리더 업스트림 운영자에게 이벤트 재생을 알립니다. (4) 업스트림 시작 이벤트를 Ndest로 재우러합니다. (5) 리더는 Nsrc에 전송이 완료되었으며 Nsrc가 슬라이스 #1을 백업 그룹으로 이동했음을 알립니다.

더 이상 결과 품질을 저하시키는 접근 방식. 또 다른 중요한 차이점은 1세대 시스템의 로드 관리 접근 방식이 공유 데이터 흐름 계획을 구성할 때 여러 쿼리 실행에 영향을 미친다는 것입니다(참조):

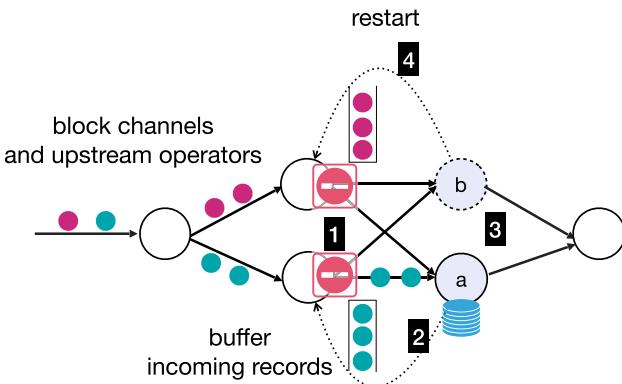
분파. 2). 최신 시스템의 쿼리는 일반적으로 독립적인 작업으로 실행되므로 특정 쿼리에 대한 역압은 동일한 클러스터에서 실행되는 다른 쿼리의 실행에 영향을 미치지 않습니다. 축소는 클라우드 배포 이전에는 문제가 되지 않았던 아주 최근의 요구사항입니다. 정확성 보장을 제공하기 위해 지속적 대기열에 대한 의존성은 주로 백프레셔를 사용하는 시스템에서 요구되는 또 다른 최근 특징입니다. 마지막으로, 초기 로드 차단 및 로드 인식 스케줄링 기술은 실행 전반에 걸쳐 속성과 특성이 안정적인 제한된 운영자 집합을 가정하는 반면, 최신 시스템은 비용과 선택성이 다양하거나 알 수 없는 경우에도 적용 가능한 일반적인 로드 관리 방법을 구현합니다.

#### 6.5 미해결 문제

적응형 스케줄링 방법은 선택성과 비용이 고정되어 있고 알려진 연산자를 사용하는 간단한 쿼리 계획의 맥락에서 지금까지 연구되었습니다. 이들 여부는 불분명하다.

메소드는 임의의 계획, UDF가 있는 연산자, 일반 창 및 사용자 정의 조인으로 일반화됩니다. 로드 인식 스케줄링은 입력 버퍼에 레코드가 있는 우선 순위가 낮은 연산자가 버스트 중에 오랜 시간을 기다려야 하므로 기아 상태를 유발하고 튜플당 대기 시간을 증가시킬 수 있습니다. 마지막으로 기존 메서드는 타임스탬프 순서로 도착하는 스트림으로 제한되며 순서가 잘못되었거나 지연된 이벤트를 지원하지 않습니다.

재구성 가능한 스트림 처리는 스트림 프로세서가 리소스 할당을 조정할 수 있을 뿐만 아니라



**Fig. 8** An example of the partial-pause-and-restart protocol. To move state from operator  $a$  to  $b$ , the mechanism executes the following steps: (1) Pause  $a$ 's upstream operators, (2) extract state from  $a$ , (3) load state into  $b$ , and (4) send a *restart* signal from  $b$  to upstream operators

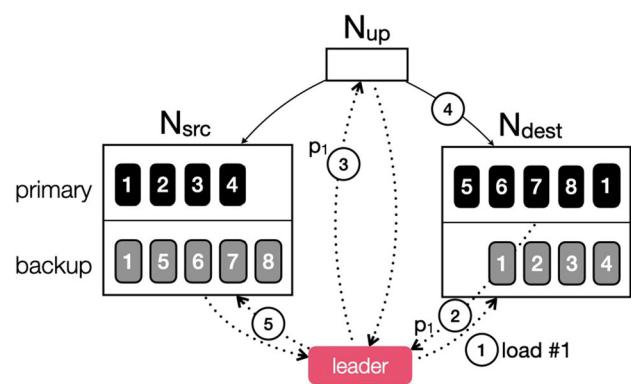
tains the operator to be scaled, as well as upstream channels and upstream operators. Figure 8 shows an example of the protocol. To migrate state from operator  $a$  to operator  $b$ , the mechanism will execute the following steps: (1) First, it *pauses*  $a$ 's upstream operators and stops pushing tuples to  $a$ . Paused operators start buffering input tuples in their local buffers. Operator  $a$  continues processing tuples in its buffers until they are empty. (2) Once  $a$ 's buffers are empty, it extracts its state and sends it to operator  $b$ . (3) Operator  $b$  loads the state and (4) sends a *restart* signal to upstream operators. Once upstream operators receive the signal they can start processing tuples again.

Systems like ChronoStream [160] and CometCloud [151] perform reconfiguration in a nearly *live* manner by leveraging a proactive replication strategy. The core idea is to maintain state backup copies in multiple nodes. To this end, state is organized into smaller partitions, each of which can be transferred independently. Nodes have a set of primary state slices and a set of secondary state slices. Figure 9 shows an example of ChronoStream's protocol.

**State transfer** Another important decision to make when migrating state from one worker to another is whether the state is moved *all-at-once* or in a *progressive* manner. If a large amount of state needs to be transferred, moving it in one operation might cause high latency during re-configuration. Alternatively, *progressive* migration [84] moves state in smaller pieces and flattens latency spikes by interleaving state transfer with processing. On the downside, progressive state migration might lead to longer migration duration.

#### 6.4 First generation versus second generation

Comparing early to modern approaches, we make the following observations. While load shedding was popular among early stream processors, modern systems do not favor the



**Fig. 9** An example of the proactive replication protocol. To move slice #1 from  $N_{src}$  to  $N_{dest}$ , the mechanism executes the following steps: (1) the leader instructs  $N_{dest}$  to load slice #1, (2)  $N_{dest}$  loads slice #1 and sends ack to the leader, (3) the leader notifies upstream operators to replay events, (4) upstream start rerouting events to  $N_{dest}$ , (5) the leader notifies  $N_{src}$  that the transfer is complete and  $N_{src}$  moves slice #1 to the backup group.

approach of degrading result quality anymore. Another important difference is that load management approaches in first-generation systems used to affect the execution of multiple queries as they formed a shared dataflow plan (cf. Sect. 2). Queries in modern systems are typically executed as independent jobs, thus, back-pressure on a certain query will not affect the execution of other queries running on the same cluster. Scaling down is a quite recent requirement that was not a matter of concern before cloud deployments. The dependence on persistent queues for providing correctness guarantees is another recent characteristic, mainly required by systems employing back-pressure. Finally, while early load shedding and load-aware scheduling techniques assume a limited set of operators whose properties and characteristics are stable throughout execution, modern systems implement general load management methods that are applicable even if cost and selectivity vary or are unknown.

#### 6.5 Open problems

Adaptive scheduling methods have been studied so far in the context of simple query plans with operators whose selectivities and costs are fixed and known. It is unclear whether these methods generalize to arbitrary plans, operators with UDFs, general windows, and custom joins. Load-aware scheduling can further cause starvation and increased per-tuple latency, as low-priority operators with records in their input buffers would need to wait a long time during bursts. Finally, existing methods are restricted to streams that arrive in timestamp order and do not support out-of-order or delayed events.

Re-configurable stream processing is a quite recent research area, where stream processors are designed to not only be capable of adjusting their resource alloca-

하지만 런타임의 다른 요소도 마찬가지입니다. 탄력, 동적으로 조정하는 스트림 프로세서의 기능 자원 할당은 특별한 경우로 간주될 수 있습니다. 재구성. 기타에는 버그 수정을 위한 코드 업데이트가 포함됩니다. 버전 업그레이드, 비즈니스 로직 변경, 실행 계획 스위칭, 동적 스케줄링 및 운영자 배치 등 왜곡 및 낙오자 완화도 포함됩니다. 지금까지 각각의 앞서 언급한 재구성 시나리오는 주로 고립되어 공부했습니다. 일반적인 재구성을 제공하려면 및 자기 관리를 위한 미래 시스템이 필요합니다. 최적화가 서로 어떻게 상호 작용하는지 설명합니다.

## 7가지 교훈과 앞으로 나아갈 길

### 7.1 설계 고려 사항에 대한 논의

이전 섹션에서 우리는 진행 상황 추적, 상태 관리, 내결합성 및 스트리밍 세대 전반에 걸친 로드 관리 시스템. 지금까지 이러한 문제 각각을 개별적으로 논의했지만 실제로는 아키텍처 결정을 내리는 경우가 많습니다. 이러한 여러 측면을 동시에 고려해야 합니다. ~ 안에 다음에서는 처리 시간, 상태, 내결합성 및 재구성에 대한 기능적 선택이 어떻게 가능한지 논의합니다. 서로 영향을 미치거나 호환되지 않게 됩니다. 이벤트 순서와 적사성을 관리하는 것은 단지 이벤트에만 영향을 미치는 것이 아닙니다. 의미론 및 결과 완전성을 보장할 뿐만 아니라 상태 관리 및 결과의 설계에도 상당한 영향을 미칠 수 있습니다. 내결합성 구성 요소. 이 연관성을 설명하기 위해 우리는 장애 관리를 위해 낮은 워터마크를 사용하는 시스템을 고려합니다. 워터마크는 본질적으로 절충안을 인코딩합니다. 대기 시간과 결과 완전성 사이에서 버퍼링해야 하는 상태의 크기에 직접적인 영향을 미칩니다. 그리고 체크포인트. 느린 워터마크는 잠재적으로 다음과 같은 결과를 초래할 수 있습니다. 더 큰 상태 크기와 더 긴 체크포인트 기간. 워터마크 전파 메커니즘의 아키텍처는 또한 복구 및 재구성 기간에 영향을 미칩니다. 외부

Google Dataflow [15] 에서와 같이 워터마크 관리를 통해 중앙 기관으로서 더 빠른 복구 및 재구성 진행 정보를 검색하기 위해 쿼리할 수 있습니다. 대조적으로, Flink와 같은 대역 내 메커니즘은 소스에서 영향을 받는 부분으로 워터마크를 전파해야 합니다.

데이터 흐름. 흥미롭게도 우리는 외부 시스템이 상태 관리는 외부 진행 상황 추적에 의존하는 경향이 있습니다. 또한 대역 내 접근 방식이 호환되지 않는 것은 아닙니다.

상태 관리에 대한 접근 방식이 내결합성 및 재구성을 최우선 고려 사항으로 설계해야 한다는 것은 놀라운 일이 아닙니다. 실제로 현대 시스템은 종종 오류에 대해 동일한 메커니즘(예: 일관된 스냅샷)을 사용합니다. 관용과 재구성. 컴퓨팅에서 상태를 분리하면 복구 및 로드 관리가 크게 단순화됩니다. 외부 상태 저장소와 컴퓨팅 리소스는 독립적으로 확장 및 구성됩니다. 마지막으로, 우리는 로드 관리 접근 방식은 결과에 직접적인 영향을 미칩니다. 의미론. 예를 들어, 로드 차단은 본질적으로 정확히 1회 보장과 호환되지 않습니다.

### 7.2 진화의 시사점

일반적인 스트림 처리 시스템 아키텍처가 발전했습니다. 지난 30년 동안 크게 증가했습니다. 초기 시스템인 반면 데이터 스트리밍을 갖춘 확장된 관계형 실행 엔진 능력을 갖추기 위해 현대 시스템의 설계가 발전했습니다. 새로운 애플리케이션 수요와 클라우드 컴퓨팅 및 하드웨어의 발전을 활용합니다. 표 6에는 주요 결과가 요약되어 있습니다. 스트림 처리 시스템의 진화에 관한 것입니다. ~ 안에 이 섹션에서 우리는 일부 이유를 밝히는 것을 목표로 합니다. 다른 접근 방식은 여전 세대에 걸쳐 지속되었지만 다른 접근 방식은 조정하거나 버려야했습니다. 초기 시스템은 종종 대략적인 결과를 반환했지만 이후 세대에서는 스트림 처리라는 개념을 거부했습니다. 근사 계산의 동의어입니다. 특히, 확장으로 시작된 스트리밍 시스템은 정확하고 정확한 제공에 초점을 맞춘 MapReduce 모델 실패하더라도 결과를 얻을 수 있습니다. 어느 정도 대략적인 프로-

표 6 스트리밍 시스템의 진화

	1세대	2~3세대
결과	대략적이거나 정확함	정확한
프로그래밍 인터페이스	SQL 확장, CQL	UDF가 많이 사용됨 - Java, Scala, Python, SQL 유사 등
쿼리 계획	전역, 최적화, 사전 정의된 연산자 포함	독립적이며 사용자 정의 연산자 사용
쿼리 실행	(주로) 규모 확장	분산
병행	관로	데이터, 파이프라인, 태스크
시간과 진행	심장박동, 느슨함, 구두점	로우 워터마크, 프론티어
상태 관리	공유 개요, 메모리 내	쿼리당, 분할되고, 지속적이며, 메모리보다 큽니다.
결합 허용	HA 중심, 제한된 정확성 보장	분산된 스냅샷, 정확히 한 번
부하 관리	로드 차단, 로드 인식 스케줄링	배압, 탄력성

tion but other elements of their runtime as well. Elasticity, the ability of a stream processor to dynamically adjust resource allocation can be considered as a special case of re-configuration. Others include code updates for bug fixes, version upgrades, or business-logic changes, execution-plan switching, dynamic scheduling and operator placement, as well as skew and straggler mitigation. So far, each of the aforementioned re-configuration scenarios have been largely studied in isolation. To provide general re-configuration and self-management, future systems will need to take into account how optimizations interact with each other.

## 7 Lessons learned and the road ahead

### 7.1 Discussion of design considerations

In the previous sections, we examined the evolution of progress tracking, state management, fault tolerance, and load management throughout the generations of streaming systems. While we have so far discussed each of these concerns in isolation, in practice, architectural decisions often have to simultaneously consider multiple of these aspects. In the following, we discuss how functional choices for handling time, state, fault tolerance, and reconfiguration can impact one another or become incompatible.

Managing event order and timeliness does not only affect semantics and result completeness, but can also have significant impact on the design of the state management and fault tolerance components. To illustrate this association, let us consider systems employing low watermarks for managing disorder. Watermarks inherently encode a trade-off between latency and result completeness, which, in turn directly affects the size of state that needs to be buffered and checkpointed. Slow watermarks can potentially lead to higher state size and longer checkpoint duration. The architecture of the watermark propagation mechanism can also affect the recovery and reconfiguration duration. External

watermark management, as in Google Dataflow [15], enables faster recovery and reconfiguration, as the central authority can be queried to retrieve progress information. In contrast, in-band mechanisms, such as the one in Flink, need to propagate watermarks from the sources to the affected parts of the dataflow. Interestingly, we observe that systems with external state management tend to rely on external progress tracking as well, though in-band approaches are not incompatible.

It comes as no surprise that the approach to state management must be designed with fault tolerance and reconfiguration as first-class concerns. In fact, modern systems often rely on the same mechanism (e.g. consistent snapshots) for fault tolerance and reconfiguration. Decoupling state from compute substantially simplifies recovery and load management, as the external state store and the compute resources can be scaled and configured independently. Finally, we note that the load management approach directly impacts the result semantics. For example, load shedding is inherently incompatible with exactly-once guarantees.

### 7.2 Evolution take-aways

The typical stream processing system architecture has evolved significantly over the last three decades. While early systems extended relational execution engines with data streaming capabilities, the design of modern systems evolved to address new application demands and exploit advances in cloud computing and hardware. Table 6 summarizes our main findings concerning the evolution of stream processing systems. In this section, we aim to shed light on the reasons why some approaches persisted throughout generations, while others had to be adjusted or abandoned.

While early systems often returned approximate results, later generations rejected the notion that stream processing is a synonym of approximate computation. In particular, streaming systems that originated as extensions of the MapReduce model focused on providing exact and correct results, even under failures. To some extent, approximate pro-

**Table 6** Evolution of streaming systems

	1st generation	2nd–3rd generation
Results	Approximate or exact	Exact
Programming interface	SQL extensions, CQL	UDF-heavy—Java, Scala, Python, SQL-like, etc.
Query plans	Global, optimized, with pre-defined operators	Independent, with custom operators
Query execution	(Mostly) scale-up	Distributed
Parallelism	Pipeline	Data, pipeline, task
Time and progress	Heartbeats, slack, punctuations	Low-watermark, frontiers
State management	Shared synopses, in-memory	Per query, partitioned, persistent, larger-than-memory
Fault tolerance	HA-focused, limited correctness guarantees	Distributed snapshots, exactly-once
Load management	Load shedding, load-aware scheduling	Backpressure, elasticity

<p>스트리밍 상태 관리는 다음과 같은 큰 변화를 목격했습니다.</p> <p>현재 지원되는 대규모 분할 및 영구 상태에 대한 전문적인 메모리 내 사냅시스입니다. 이 변화는 어느 정도 대략적인 계산에서 정확한 계산으로, 중앙 집중식에서 분산 및 클라우드 기반으로 전환한 결과</p> <p>배포. 결과적으로 내결합성 및 고가용성도 수동 복제 및 정확히 한 번 복제로 전환되었습니다.</p> <p>처리. 내결합성과 높은 수준에 대한 다양한 접근 방식</p>	<p>현재 사용 중인 가용성(예: 활성 및 수동) 복제 및 업스트림 백업은 이미 1세대 시스템. 그러나 이후 세대에서는 세련되어 이러한 기술을 확장하여 정확히 한 번 의미론을 보장했습니다.</p> <p>상태 관리에서 우리는 가장 급진적인 변화를 식별합니다. 지금까지 데이터 스트리밍에서 볼 수 있었습니다. 가장 확실한 발전 상태의 확장성과 장기적인 지속성과 관련됩니다.</p> <p>무제한 실행에서. 오늘날의 시스템은 투자했습니다. 철저하게 거래 보증을 제공합니다.</p> <p>현재 데이터베이스 관리 시스템과 동등합니다.</p> <p>트랜잭션 스트림 처리는 데이터 스트리밍을 중심으로 했습니다.</p> <p>데이터 분석 사용 사례를 넘어서 새로운 기능도 열었습니다.</p> <p>효율적인 지원 방법에 관한 연구 방향</p> <p>무제한으로 성장하는 상태에 액세스합니다. 개울 상태와 컴퓨팅은 점차 분리되고 있으며 이는 추세는 더 나은 최적화, 스토리지 기술과 더 넓은 상호 운용성 및 새로운 의미론을 허용합니다.</p> <p>공유 및 외부 상태.</p> <h3>7.3 미래 전망</h3> <p>이번 설문조사 전반에 걸쳐 자세히 설명했듯이 데이터 환경은 스트리밍 시스템은 최근 몇 년 동안 크게 변화했습니다.</p> <p>그러나 데이터 스트리밍 시스템의 앞길은 여전히 멀고 변화로 가득 차 있습니다.</p> <p>진행중인 목록을 나열합니다.</p> <p>주요 카테고리의 스트림 처리의 미래 동향</p> <p>: 서비스/클라우드, 쿼리 기능, 엣지 및 하드웨어 가속.</p> <p>서비스 및 클라우드 서비스 컴퓨팅의 출현으로 처리 및 클라우드에 대한 새로운 기회가 도입되었습니다.</p> <p>뛰어난 유연성과 비용 효율성으로 데이터 스트림을 분석합니다.</p> <p>그러나 이는 또한 다음과 같은 새로운 과제를 야기합니다.</p> <p>상태 저장 작업 처리, 조정 및 관리</p> <p>변동성이 매우 높은 분산 환경의 리소스</p> <p>고유한 문제에 직면하여 저지연 처리를 보장합니다.</p> <p>서비스 플랫폼의 예측 불가능성. 데이터 스트리밍 애플리케이션이 점점 더 서비스 아키텍처를 채택함에 따라 이러한 과제를 해결하는 것이 기술을 활용하는 데 중요해집니다.</p> <p>실시간 데이터 처리 및 분석을 위한 서비스 컴퓨팅의 잠재력을 최대한 활용하세요. 스트림 일정을 조정하는 기능</p> <p>작업은 서비스 및 클라우드에서도 특히 유망합니다.</p> <p>가상화에는 런타임 적응 기능이 필요한 배포가 있습니다 [ 110 ]. Lachesis [ 127 ]는 다음의 한 예이다.</p> <p>런타임에 스트림 프로그램을 예약하도록 설계된 미들웨어입니다.</p> <p>Meces [ 74 ]는 자주 사용하는 효과적인 방법을 제시합니다.</p> <p>스트림 처리 파이프라인의 동적 재구성을 통해 상태를 마이그레이션하는 반면, Xu et.al [ 162 ]도 다음을 조사합니다.</p> <p>다중 테넌트 환경에서 SLA를 유지할 가능성에 있습니다.</p> <p>스트리밍 시스템에 대한 서비스 및 클라우드 적응에 대한 향후 연구는 스트리밍 시스템에 대한 더 깊은 이해를 얻는데 필수적입니다.</p> <p>상태 관리와 관련된 성능 균형</p>
--	---

cessing was a necessity in early systems that were deployed on restricted resources. By carefully dropping some events or emitting early incomplete results, these systems achieved high availability and low latency. In contrast, streaming systems that were designed for cloud environments could leverage the capability of adjusting their resource requirements according to the workload. As a result, these systems persist or redistribute excess load to avoid data loss and guarantee exact outputs. Despite differences in the reaction mechanism, we emphasize that the problem of detecting and quantifying overload is fundamental in both early and modern systems and it has been consistently addressed with continuous monitoring and feedback control.

In terms of programming interface, we observe a full circle. As first-generation streaming systems evolved from database management systems, the first programming interfaces for data stream queries were designed around SQL-like languages. On the other hand, second-generation systems are UDF-centric and favor general-purpose programming languages, such as Java, Scala, and Python. However, as stream processing tools are becoming widespread, we witness a trend to return to extensions for streaming SQL [32] to accommodate a larger variety of users and use cases.

Over the years, the design of streaming query execution engines has also gradually transitioned from mainly centralized to mainly distributed, exploiting data, pipeline, and task parallelism. Most modern streaming systems target shared-nothing in-house or on-cloud clusters. This shift has also impacted architectural decisions in query scheduling, optimization, and deployment. 1st-generation systems commonly share resources among multiple queries that are jointly optimized and executed. On the contrary, modern systems provide per-query resource allocation and guide users to develop and deploy independent applications, even if they ingest events from common sources. While this prevailing approach may lead to higher resource requirements and redundant computation, it offers more flexibility. Separate query deployments enable faster and easier fault recovery and reconfiguration.

Regarding time, order, and progress, many of the inventions of the past proved to have survived the test of time, since they continue to hold a place in modern streaming systems. In particular, Millwheel and the Google Dataflow Model popularized punctuations, watermarks, the out-of-order architecture, and triggers for revision processing. Streaming state management witnessed a major shift, from specialized in-memory synopses to large partitioned and persistent state supported today. To some extent, this change is a consequence of shifting from approximate to exact computation and from centralized to distributed and cloud-based deployments. As a result, fault tolerance and high availability also shifted towards passive replication and exactly-once processing. Many approaches to fault tolerance and high

availability that are in use today, such as active and passive replication and upstream backup, were already proposed in 1st-generation systems. However, later generations refined these techniques and extended them to guarantee exactly-once semantics.

In state management, we identify the most radical changes seen in data streaming so far. The most obvious advances relate to the scalability of state and long-term persistence in unbounded executions. Today's systems have invested thoroughly in providing transactional guarantees that are in par with those of current database management systems. Transactional stream processing has pivoted data streaming beyond data analytics use cases and has also opened new research directions in terms of efficient methods for backing and accessing state that grows in unbounded terms. Stream state and compute are gradually being decoupled, and this trend allows for better optimizations, wider interoperability with storage technologies as well as novel semantics for shared and external state.

### 7.3 A future outlook

As we detailed throughout this survey, the landscape of data streaming systems has changed significantly in recent years. Yet, the road ahead for data streaming systems is still evidently long and full of transformations. We list out ongoing and future trends in stream processing in the key categories of: serverless/cloud, query capabilities, edge and hardware acceleration.

**Serverless and cloud** The advent of serverless computing has introduced new opportunities for processing and analyzing data streams with greater flexibility and cost efficiency. However, it also brings forth new challenges, such as handling stateful operations, orchestrating and managing resources in a highly volatile distributed environment, and ensuring low-latency processing in the face of the inherent unpredictability of serverless platforms. As data streaming applications increasingly adopt serverless architectures, addressing these challenges becomes crucial to harness the full potential of serverless computing for real-time data processing and analytics. The ability to tune scheduling of stream tasks is also particularly promising in serverless and cloud deployments, where virtualization necessitates runtime adaptation capabilities [110]. Lachesis [127] is one example of a middleware designed for scheduling stream programs at runtime. Meces [74] presents an effective method to frequently migrate state across dynamic reconfigurations of stream processing pipelines, whereas, Xu et.al [162] also investigate the prospect of maintaining SLAs in multi-tenant environments. Future research in serverless and cloud adaptation for streaming systems is essential to gain a deeper understanding of the performance trade-offs associated with state management

접근 방식뿐만 아니라 재구성 전략을 탐구합니다.

가상화되고 고도로 동적인 인프라의 맥락.

쿼리 기능 Stateful 채택이 증가함에 따라

스트림 엔진, 최근 연구 초점은 로컬 전용 데이터 흐름을 넘어 상태 저장 기능의 향상을 목표로 합니다.

주에 대한 접근. S-Query [159]는 외부 쿼리에 대한 연산자 상태 노출 가능성을 조사합니다.

격리 수준이 다릅니다. 스냅샷 상태는 스냅샷 격리를 사용하여 읽기 전용 쿼리를 활성화합니다. 스트림의 출처

쿼리는 스트림 처리의 또 다른 새로운 측면입니다.

출처에는 설명 가능성 및 스트림 실행과 같이 특히 흥미로운 여러 용도가 있습니다.

스트리밍 쿼리 프로파일링. Ananke [126] 는 출처 실행 전략의 한 예입니다. 향후 연구

쿼리 기능은 SQL 통합(예: 스트림 및 테이블 [32])의 과제를 해결해야 합니다.

사용자가 스트림과 상호 작용할 수 있는 방법에 대한 새로운 표준 개발 제한된 데이터 흐름 모델을 넘어서는 파이프라인.

상태 저장 서비스 기능을 위한 프로그래밍 모델

현재 스트리밍 데이터 흐름 시스템은 기능적 프로그래밍 스타일 데이터 흐름을 통해서만 프로그래밍 가능합니다.

아피스. 이에 따라 결제처리, 예약시스템, 재고관리,

지연 시간이 짧은 비즈니스 워크플로를 다시 작성해야 합니다.

이벤트 기반 데이터 흐름 패러다임과 일치하는 프로그래머.

이 방법으로 많은 응용 프로그램을 다시 작성할 수는 있지만

패러다임에서는 상당한 양의 프로그래머가 필요합니다.

그리기 위한 훈련과 노력. 우리는 스트리밍 데이터 흐름이

시스템은 새로운 프로그래밍 추상화의 이점을 누릴 수 있습니다.

[128] 프로그래머가 일반 용도로 채택하기 위해

클라우드 애플리케이션.

트랜잭션 서비스 기능은

스트리밍 위에서 실행되는 상태 저장 기능 간에 트랜잭션 조정을 도입하는 것이 가능하다는 것을 보여주었습니다.

데이터 흐름 시스템 [60], 우리는 트랜잭션을 보다 효율적으로 구현하면서 데이터 흐름 시스템이 필요하다고 주장합니다.

거래 경계를 인식하고 통합하기 위해

트랜잭션을 상태 관리 및 내결합성 프로토콜로 처리합니다. 사용하는 가장 중요한 이유

일반 클라우드 애플리케이션을 위한 데이터 흐름 시스템은 최신 스트리밍 데이터 흐름 시스템이 메시지 처리 기능을 제공한다는 것입니다.

보장합니다(정확히 한 번만 처리). 결과적으로 프로그래머는 자신의 비즈니스 로직을 다음과 같이 "오염" 시킬 필요가 없습니다.

일관성 검사, 상태 롤백, 시간 초과, 재시도 및

멱등성 [94, 103].

엣지 스트리밍 5G와 호환 엣지의 등장

하드웨어는 데이터 스트림 이동에 큰 관심을 불러일으켰습니다.

에지 노드로의 파이프라인(예: 센서 장치, 웨어러블 기기,

모바일, 기지국 등). 공정을 더 가까운 곳으로 이동

소스는 타의 추종을 불허하는 낮은 처리 지연 시간과

전처리 비용 절감 효과를 창출하세요. 두 가지 주요 과제

이 설정에서 해결되는 것은 로우엔드 IoT입니다.

하드웨어 실행 및 데이터 스트리밍 파이프라인의 분산화. Edgewise [68]는 새로운 스케줄링을 제안합니다.

IoT 노드에서 작업을 스트리밍하기 위한 방법론

무거운 리소스 없이 더 많은 리소스 경량 처리 및 IO

성능의 타협. 마찬가지로 헤이즐캐스트 제트(Hazelcast Jet) [72]

과도한 지속 상태 및 트랜잭션 비용을 제거합니다.

낮은 대기 시간과 인메모리에 대한 절충을 보장합니다.

처리 포털 [140, 141]은 사전 정의된 사용을 제한합니다.

스트림을 원자 단위로 나누어 트랜잭션 처리

"원자"라고 불리는 세그먼트. 포털의 Atom은 서로 다른 스트림 파이프라인을 교차할 때 트랜잭션 실행을 계획합니다.

엣지와 클라우드는 데이터 통신이 필요합니다. 미래 연구

인 엣지 스트리밍은 계속해서 경량화를 연구할 예정입니다.

기술 및 데이터 스트리밍 시스템과의 통합

리소스가 제한된 환경에서 효율적이고 지연 시간이 짧은 처리를 가능하게 합니다.

하드웨어 가속 스트림에서 최적화된 코드 생성

현재 GPU 및 FPGA 형태의 가속기 채택이 증가함에 따라 처리가 증가하고 있습니다.

향상된 메모리 및 스토리지의 확산

NVME와 같은 기술. 진행중인 사례 중,

Brisk [166]은 공유 메모리 멀티코어 아키텍처의 스트림 확장성을 보여줍니다.

FineStream [165] 향상

CPU-GPU의 윈도우 기본 실행 성능

통합 아키텍처. GPU와 인터페이스 할 때 일반적인 과제는 사용자 지정 드라이버 침을 작성해야 한다는 것입니다.

특정 스트림 운영자의 경우. 하나의 실천적 방향

하드웨어 가속은 자동화된 GPU 실행을 직접적으로 수행합니다.

JVM 관련 번역된 바이트코드 코드 조각에서

스트리밍 프레임워크(예: Flink)

Tornado/GraalVM 프로젝트 [97]. 사용할 전망

데이터 스트리밍을 위한 FPGA는 다음에서도 시연되었습니다.

함대 프로젝트 [149]. 향후 연구는 탐구가 필요하다

다음과 같은 새로운 하드웨어 기술의 잠재력

RISC-V 명령어 세트와 새로운 패러다임. NMC(Near-Memory Computing) [139]는 이러한 패러다임 중 하나입니다.

특히 스트림 처리에 매력적입니다.

컴퓨팅 작업을 해당 주소와 함께 배치

동일한 칩에 공간을 확보하여 Von-Neumann을 방지합니다.

RAM과 CPU 사이의 데이터 전송 병목 현상

스트리밍 애플리케이션에서 일반적입니다.

감사의 말 익명의 VLDBJ 검토자에게 감사드립니다.

이 백서의 이전 초안에 대한 상세하고 귀중한 피드백. 이것

이 작업은 Google DAPA 상인 WASP NESTS에서 부분적으로 지원되었습니다.

(Data-Bound Computing) 및 네덜란드 연구 위원회(NWO)

프로젝트 번호를 참조하세요. 19708.

오픈 액세스(Open Access) 이 기사는 크리에이티브 커먼즈(Creative Commons)에 따라 라이센스가 부여됩니다.  
사용, 공유, 조정을 허용하는 Attribution 4.0 국제 라이선스

approaches, as well as to explore reconfiguration strategies in the context of virtualized and highly dynamic infrastructures.

**Query capabilities** With the increased adoption of stateful stream engines, a recent research focus targets the enhancement of the stateful capabilities beyond local-only dataflow access to the states. S-Query [159] examines the possibility of exposing operator states for external queries with different isolation levels. Snapshotted state enables read-only queries using snapshot isolation. Provenance in stream queries is another emergent aspect of stream processing. Provenance has multiple uses that are particularly interesting in stream execution, such as explainability as well as profiling of streaming queries. Ananke [126] is one example of a provenance execution strategy. Future research on query capabilities will further need to address the challenges of SQL integration (e.g., streams and tables [32]) and develop new standards in how users can interact with stream pipelines, beyond the restricted dataflow model.

### Programming models for stateful serverless functions

Streaming dataflow systems at the moment are only programmable through functional-programming style dataflow APIs. As a result, general cloud applications such as payment processing, reservation systems, inventory keeping, and low-latency business workflows need to be rewritten by programmers to match the event-driven dataflow paradigm. Although it is possible to rewrite lots of applications in this paradigm, it takes a considerable amount of programmer training and effort to do so. We argue that streaming dataflow systems would benefit from new programming abstractions [128], for them to be adopted by programmers for general cloud applications.

**Transactional serverless functions** Although it has been shown that it is possible to introduce transaction coordination among stateful functions running on top of a streaming dataflow system [60], we argue that a more efficient implementation of transactions would require the dataflow system to be aware of transaction boundaries and to incorporate transaction processing into its state management and fault-tolerance protocols. The most important reason for using dataflow systems for general cloud applications, is that modern streaming dataflow systems offer message processing guarantees (exactly-once processing). As a result, programmers do not need to “pollute” their business logic with consistency checks, state rollbacks, timeouts, retries, and idempotency [94, 103].

**Edge streaming** The emergence of 5 G and compatible edge hardware has created great interest in moving data stream pipelines to edge nodes (e.g., sensor devices, wearables,

mobiles, base-stations etc.). Moving processing closer to the sources can lead to unmatched low processing-latency and create pre-processing cost savings. Two major challenges that are being addressed in this setting are low-end IoT hardware execution and the decentralization of data streaming pipelines. Edgewise [68] proposes a new scheduling methodology for streaming tasks on IoT nodes employing a more resource-lightweight processing and IO without heavy compromises in performance. Similarly, Hazelcast Jet [72] removes the cost of heavy persistent state and transactional guarantees to compromise for low-latency and in-memory processing. Portals [140, 141] proposes the use of pre-defined transactional processing by dividing streams into atomic segments called “atoms”. Atoms in portals instrument transactional execution when different stream pipelines across edge and cloud need to communicate data. Future research in edge streaming will continue to investigate lightweight techniques and their integration with data streaming systems to enable efficient and low-latency processing in resource-constrained environments.

**Hardware acceleration** Optimized code generation in stream processing is currently on the rise due to the increased adoption of accelerators in the form of GPUs and FPGAs, as well as the proliferation of enhanced memory and storage technologies, such as NVMEs. Among ongoing examples, Brisk [166] is a showcase of stream scalability on shared-memory multicore architectures. FineStream [165] enhances the performance of window-based execution for CPU-GPU integrated architectures. A typical challenge when interfacing with GPUs is the need to write custom driver instructions for specific stream operators. One practical direction in hardware acceleration is automated GPU execution directly from translated bytecode code fragments in JVM-specific streaming frameworks (e.g., Flink) as demonstrated in the Tornado/GraalVM projects [97]. The prospect of using FPGAs for data streaming has also been demonstrated in the Fleet project [149]. Future research will need to explore the potential of emerging hardware technologies, such as RISC-V instruction sets, as well as new paradigms. Near-Memory Computing (NMC) [139] is one such paradigm that is particularly attractive for stream processing, which aims to co-locate computing tasks with their corresponding address space in the same chip, thereby, avoiding the Von-Neumann data transfer bottleneck between RAM and CPU, which is common in streaming applications.

**Acknowledgements** We thank the anonymous VLDBJ reviewers for their detailed and valuable feedback on prior drafts of this paper. This work was partially supported by a Google DAPA award, WASP NESTS (Data-Bound Computing), and the Dutch Research Council (NWO) Vidi project No. 19708.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adap-