Streams and Tables: Two Sides of the Same Coin

Matthias J. Sax*
Confluent Inc.
Palo Alto, USA
matthias@confluent.io

Guozhang Wang
Confluent Inc.
Palo Alto, USA
guozhang@confluent.io

Matthias Weidlich
Humboldt-Universität zu Berlin
Berlin, Germany
matthias.weidlich@hu-berlin.de

Johann-Christoph Freytag
Humboldt-Universität zu Berlin
Berlin, Germany
freytag@informatik.hu-berlin.de

ABSTRACT

Stream processing has emerged as a paradigm for applications that require low-latency evaluation of operators over unbounded sequences of data. Defining the semantics of stream processing is challenging in the presence of distributed data sources. The physical and logical order of data in a stream may become inconsistent in such a setting. Existing models either neglect these inconsistencies or handle them by means of data buffering and reordering techniques, thereby compromising processing latency.

In this paper, we introduce the Dual Streaming Model to reason about physical and logical order in data stream processing. This model presents the result of an operator as a stream of successive updates, which induces a duality of results and streams. As such, it provides a natural way to cope with inconsistencies between the physical and logical order of streaming data in a continuous manner, without explicit buffering and reordering. We further discuss the trade-offs and challenges faced when implementing this model in terms of correctness, latency, and processing cost. A case study based on Apache Kafka illustrates the effectiveness of our model in the light of real-world requirements.

CCS CONCEPTS

• **Information systems** → **Data streams**; *Stream management*;

KEYWORDS

Stream Processing, Processing Model, Semantics

ACM Reference Format:
Matthias J. Sax, Guozhang Wang, Matthias Weidlich, and Johann-Christoph Freytag. 2018. Streams and Tables: Two Sides of the Same Coin. In *International Workshop on Real-Time Business Intelligence and Analytics (BIRTE '18), August 27, 2018, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3242153.3242155

1 INTRODUCTION

Stream processing has emerged as a paradigm to develop real-time applications. It builds on an evaluation of operators over unbounded sequences of data, enabling low-latency processing of large-scale data in a continuous manner [11]. As such, the stream processing paradigm turned out to be particularly suited to support the implementation of business logic in large organizations. It provides the backbone for communication between independent components of a large system, a. k. a. "microservices", through asynchronous message-passing [19].

Defining semantics for stream processing, however, is demanding. One needs to cope with challenges imposed by the nature of data streams, the expressiveness of operators, and trade-offs faced when implementing a streaming model:

Data-related challenges: Data sources of contemporary streaming applications are inherently distributed (e. g., in IoT use cases) and commonly assign timestamps to streaming data, which induces a logical order. Yet, the physical order of data arriving at a stream processing system may be inconsistent with this logical order [25], due to imperfect clock synchronization, network delays, or sources buffering data while being disconnected for some time [4] (e. g., a mobile phone pushes data produced during a flight after reconnecting to the network). Hence, a stream processing model must be able to handle out-of-order data arrival.

Operator-related challenges: A stream processing model shall avoid implicit operator semantics, in order to achieve

*Additional affiliation: HU Berlin, mjsax@informatik.hu-berlin.de

BIRTE '18, August 27, 2018, Rio de Janeiro, Brazil

ACM ISBN 978-1-4503-6607-6/18/08.
https://doi.org/10.1145/3242153.3242155

deterministic and well-defined processing results [12]. In particular, operators may be stateful and need to be based on the timestamps assigned to data in a stream, i. e., the logical order of streaming data [6]. Furthermore, the infinite nature of data streams implies a trade-off between processing cost, latency, and result completeness. Modelling this trade-off explicitly and in the light of operator properties (e. g., distributive, algebraic, and holistic functions [13]) allows users to reason about the system behavior.

System-related challenges: The idea of stream processing is online handling of data, so that a low processing latency is of utmost importance. Hence, when implementing a stream processing model, one cannot rely on centralised algorithms in order to enable distributed evaluation of operators. At the same time, data buffering (e. g., for reordering data to resolve inconsistencies of logical and physical ordering) [1, 27] shall be avoided, as it would compromise processing latency [17]. With records that are potentially delayed by hours (e. g., disconnected mobile phone during a flight) decoupling processing latency from handling out-of-order data is paramount.

Over the past decades, a plethora of stream processing models have been presented in the literature. Yet, we argue that existing models target only a subset of the aforementioned challenges. For instance, the seminal work on the Continuous Query Language (CQL) [5] provides well-defined semantics for relational operators, but neglects issues stemming from out-of-order data. Other models, in turn, focus on handling of unordered data, but fail to address some of the other related challenges: stream punctuations [27] lead to increased processing latency; the "time-travel" mechanism of Borealis [2] stores the stream history, blurring the idea of online data handling; while existing models for update/delete data streams [7] lack formal semantics for operator or do not address the challenges of distributed processing [17, 21]. As such, we focus on the question of *how to design a model for the evaluation of expressive operators over potentially unordered data streams that can be implemented by means of distributed, online algorithms.*

In this work, we present the Dual Streaming Model to address the above question and to unify existing approaches in a holistic model. The main idea of this model is to represent the result of an operator, captured by the relational notion of a table, as a stream of successive updates. This induces a duality of tables and streams, that enables reasoning over inconsistencies between the physical and logical order of streaming data.

More specifically, our contributions as well as the structure of the paper, following background on data streams in the next section, are summarised as follows:

- We present the Dual Streaming Model (Section 3). It puts forward a duality of streams and tables (representing operator results). The latter is modelled by a changelog stream, which enables reasoning over inconsistencies in the logical and physical order of data.
- We show how this model enables the definition of semantics for a wide range of streaming operators (Section 4). We also highlight how our novel model makes explicit implementation trade-offs in terms of correctness, latency, and processing cost.
- We demonstrate the effectiveness of our model (Section 5). To this end, we report on a case study, addressing real-world requirements using an implementation of our model in Apache Kafka[1].

Finally, we review our contributions in the light of related work (Section 6), before we conclude the paper (Section 7).

2 BACKGROUND

This section reviews essential notions of data stream processing, in terms of a data model and operators.

2.1 Data Streams

A *data stream* is an append-only sequence of immutable *records* [1, 6]. We adopt a model in which each record comprises an offset, a timestamp, and some payload.

The *offset* of a record is its unique sequence number, assigned when appended to a data stream (potentially by some middleware layer). Records produced by multiple sources are appended in interleaved order according to their arrival order, such that offsets, captured by natural numbers, define a *strict* total order over all records of a data stream.

Each record has a *timestamp*, assigned by its source (sometimes called event-time or application-time). Following [8, 25], we model time by natural numbers. Multiple records can have the same timestamp. Yet, the number of records with the same timestamp is finite.

To capture the *payload* of a record, we adopt a model based on *key-value-pairs*. The optional key hereby identifies the particular data that is carried by the record, which is then captured by its value. We denote the domains of keys and values by $\mathbb{D}_K$ and $\mathbb{D}_V$, respectively. Values may be given, for instance, as scalars, composite data types such as relational tuples, or nested data types such as those based on XML.

Formally, we model a record r as a tuple $\langle o, t, k, v\rangle \in \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{D}_K \times \mathbb{D}_V$. Record offset, timestamp, key, and value are denoted as $r.o$, $r.t$, $r.k$, and $r.v$.

A data stream is denoted by $S = (r_0, r_1, r_2, \dots)$ with r_i being a record with $r_i.o = i$. All records of a stream must have the same *schema* $\mathbb{S} = \mathbb{D}_K \times \mathbb{D}_V$, also referred to as the schema of the stream. A data stream S with schema $\mathbb{S}$ is denoted as $S[\mathbb{S}]$ and the domain of all such streams is $S[\![\mathbb{S}]\!]$.

[1] https://kafka.apache.org/

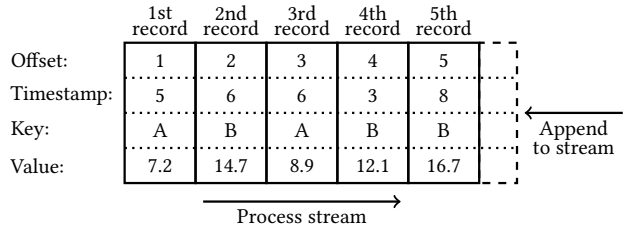

Figure 1: Example data stream with five records.

Data streams are created by appending records to an initially empty sequence. Let $S = (r_0, \ldots, r_n)$ be a data stream with schema $\mathbb{S}$. Given a tuple $\langle t, k, v \rangle$ of a timestamp and a key-value pair according to $\mathbb{S}$, a new record is appended to S, which yields the stream $S' = (r_0, \ldots, r_n, r_{n+1})$ with $r_{n+1} = \langle n+1, r.t, r.k, r.v \rangle$.

The essential notions of our model are illustrated in Figure 1. The respective data stream consists of five records, with $\mathbb{D}_K = \{A, B\}$ as the domain of keys and $\mathbb{D}_V = \mathbb{Q}$ as the domain of values. For instance, the second record in the stream is given as $\langle 2, 6, B, 14.7 \rangle$. Note that records are enumerated by their offsets.

Our model defines a *physical order* and a *logical order* of records in a data stream. The former, a strict total order, is given by the offsets (hence, also called offset order) and represents the order in which records have been appended to a stream. The logical order, in turn, is induced by the timestamps of the records. Since timestamps, in contrast to offsets, are not necessarily unique, we use the record offset as "tie breaker" [15] to derive a logical order that is *strict* and *total* over all records.

Definition 2.1 (Logical Record Order). Let r and r' be records of a data stream S. Record r is *earlier than (or before)* record r', denoted as $r < r'$, iff:

$$r.t < r'.t \vee (r.t = r'.t \wedge r.o < r'.o)$$

We also say that r' is *later than (or after)* r.

The physical order, in which records are appended to a stream, may be inconsistent with their logical order [6] (c. f. Figure 1). In practice, such out-of-order data is very common and results from imperfect clock synchronization, network delays, or data buffering, in particular in distributed applications [4]. We call streams that contain out-of-order data *unordered* streams in contrast to *ordered* streams.

2.2 Streaming Operators

The above model of append-only data streams implicitly defines the physical storage and processing model. Data is stored in offset order and processing is performed as a *single linear scan* over the input data streams. This is important, because it implies that records are *processed in physical order*, i. e., in the order in which they are appended to the stream.

Following the literature, we consider stateless operators like `filter`, `projection`, `map`, and `flatMap` [1]. They are applied to each record of an input stream, while the results are appended to an output stream (unlike `map`, `flatMap` produces none, one, or more output records per input record). As the operators are applied in offset order, they preserve the physical order: Records are appended to the output stream in the same order as the originating records were appended to the input stream.

Additionally, we consider stateful operators like aggregation (`agg`) and `join`. Stateful operators are more expressive than stateless operators because their computational logic can consider more than a single record during processing. The state is used to either "remember" previous records or to maintain a (partial) result. For each input record and based on the current state, the operator may produce output records and/or update its state. For instance, an aggregation that *counts* records would have as state the current count; for each processed record the operator increments the count by one and updates its state accordingly.

3 DUALITY OF STREAMS AND TABLES

In most existing streaming models, operators directly yield an output data stream and the treatment of operator state is considered to be an implementation detail. That is, models such as CQL [5] ignore how inconsistencies of the physical and logical order of a stream may influence the computation of operator state and, therefore, the resulting output stream. Out-of-order records are neglected in the stream processing model and need to be handled on a technical layer. In practice, this requires that the evaluation of operators is delayed until the logical order of records has been established. As a consequence, the latency with which the output stream is computed grows linear with the "maximum lateness" of records [17, 24], i. e., the difference between the timestamp of an out-of-order record r and the largest timestamp of all records with a smaller offset than r's offset. Thus, latency is dominated by the characteristics of the data stream instead of the semantics and implementation of the operator.

To overcome such dependency, we propose a model in which the operator result is *updated continuously* [8, 18]. This enables us to drop any assumption on the consistency of the logical and physical order of records. Furthermore, the result of an operator may be viewed either statically, as its materialization from processing a stream up to an offset, or dynamically, as a stream of successive updates. Both views induce just two ways of programming with respect to *time*: One may process a stream of result updates or continuously query the materialized result to achieve the same computational logic.

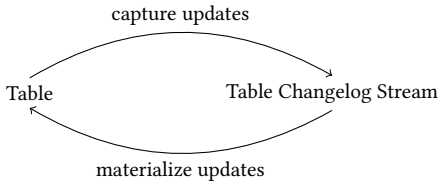

Figure 2: Duality of streams and tables.

The above idea is realised in our *Dual Streaming Model*, which comprises the notions of a *table*, a *table changelog stream*, and a *record stream*.

The static view on the current result of an operator is given by a *table*, which is updated for each processed input record. In line with our model as defined in Section 2, a table has a primary key attribute. For instance, for an agg operator, grouping conditions (e. g., time windows or group-by clauses) define the primary key of a table. If an out-of-order record is processed, the table is updated for the respective key similar to processing an in-order record. Hence, handling out-of-order records does not increase the latency of evaluating the operator: Updating an "older" entry is not more expensive than updating a "recent" one, because both is a key-value lookup in a hash table.

The dynamic view on the result of an operator is a *table changelog stream*, which consists of records that are updates to a table. The semantics of an update is defined over record keys and timestamps, i. e., a later record with a particular key updates an earlier record with the same key. Thus, in a table changelog stream, record keys have primary key semantics and replaying a table changelog stream allows to materialize the operator result as a table. We call the resulting relation the *duality of streams and tables* (Figure 2).

Applying a table changelog stream to an initially empty table, yields a table with the *latest* result per key. However, for stream processing operators that have temporal semantics, we need to reason about the content of a table over time. Hence, we consider tables to maintain multiple *versions* at once: A table is a collection of table versions; one version at each point in time using the timestamp as version number. A table version with number t is denoted by T_t. While input data is processed, time progresses and a table evolves: new table versions are added and older table versions might be updated if out-of-order data is processed. To reason about the state of a table, we assign a *generation* number to it. For each processed input record, the generation is incremented by one. Thus, a table's generation is the same as the offset of the latest processed input record. We denote a table T of generation o as $T(o)$. Note that each table generation may consist of multiple table versions.

Example 3.1 (Evolving table). Let the stream from Figure 1 be a table changelog stream. Materializing the stream evolves a table from generation 1 to 5, incorporating table versions T_3, T_5, T_6, T_8 (updates are highlighted):

$$T(1) = \{T_5 = \{\langle A, 7.2\rangle\}\}$$
$$T(2) = \{T_5 = \{\langle A, 7.2\rangle\}, T_6 = \{\langle B, 14.7\rangle\}\}$$
$$T(3) = \{T_5 = \{\langle A, 7.2\rangle\}, T_6 = \{\langle A, 8.9\rangle, \langle B, 14.7\rangle\}\}$$
$$T(4) = \{T_3 = \{\langle B, 12.1\rangle\}, T_5 = \{\langle A, 7.2\rangle, \langle B, 12.1\rangle\}, T_6 = \{\langle A, 8.9\rangle, \langle B, 14.7\rangle\}\}$$
$$T(5) = \{T_3 = \{\langle B, 12.1\rangle\}, T_5 = \{\langle A, 7.2\rangle, \langle B, 12.1\rangle\}, T_6 = \{\langle A, 8.9\rangle, \langle B, 14.7\rangle\}, T_8 = \{\langle A, 8.9\rangle, \langle B, 16.7\rangle\}\}$$

Note, that going from generation 3 to 4 an out-of-order record is processed. Therefore, not only a new table version 3 is created but also T_5 is updated.

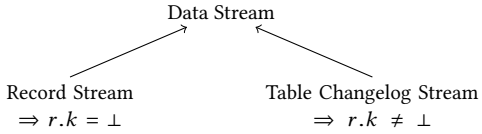

Figure 3: Data stream types and their relationship.

Finally, our model comprises *record streams* in which records represent facts (and not updates, as in table changelog stream). For example, a record stream could be used to capture a click stream for web page monitoring: Each click by a single user is a captured by a record in the stream.

Record streams and table changelog streams denote special cases of data streams as introduced in Section 2, see Figure 3. For record streams, which model immutable facts, each record must have a unique key over the whole stream. We introduce a special key $\perp$ with $\perp \neq \perp$ and use it to denote the unique keys of records in a record stream ($r.k = \perp$). For table changelog streams, we assume that each record has a key, i. e., $r.k \neq \perp$, and that the number of unique keys is finite. The latter means that a table materialized from a table changelog stream is finite.

4 STREAM PROCESSING OPERATORS

In this section, we introduce the stream processing operators of our model. We distinguish different types of operators depending on their properties. Operators may be stateless or stateful, have one or multiple input streams, and may be defined on special types of input streams only. Figure 4 summarizes all transformations between different types of streams/tables for different operators. Record streams are transformed into record streams (Section 4.1) with the exception of the aggregation operator that yields a result table (Section 4.2). Tables are always transformed into new tables (Section 4.3), unless joined with a stream, in which case the result is an output stream (Section 4.4). Furthermore, as discussed in Section 3, tables and table changelog streams can be converted into each other.

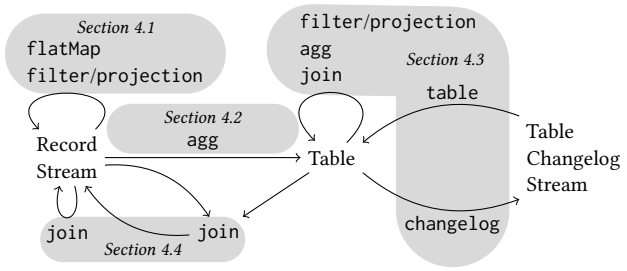

Figure 4: Transformations between record streams, tables, and table changelog streams.

To abstract over ordered and unordered input streams, we define *stream equivalence* for record streams and table changelog streams. Ordered streams are canonical representatives of an equivalence class of streams that contains both, ordered and unordered streams. Hence, we only provide operator definitions based on ordered input streams. Exploiting stream equivalence, we extend the original definition for unordered input streams, without redefining the operator.

We motivate our definition for *stream equivalence* for record streams using the example of a record input stream and a filter operator. Note that records have $k = \perp$ for this case and thus we omit the key in the following discussion.

Example 4.1 (Filtering ordered/unordered record streams). Given an ordered and an unordered record stream $S[\mathbb{S}]$ and $S'[\mathbb{S}]$ that contain the same records r, s, t, v. Let a `filter` operator applied to each stream yield the following result:

$$\texttt{filter}(S) = \texttt{filter}((r, s, t, v)) = (r, t, v)$$

$$\texttt{filter}(S') = \texttt{filter}((r, s, v, t)) = (r, v, t)$$

Both input streams and both output streams contain the same data (i. e., timestamps and values), however, they are not *equal* because the records appear in different order. However, for record streams only timestamp and value should be considered to compare two streams. Offsets that describe the physical order should be ignored.

If two record streams contain the same data (i. e., timestamps and values), we consider them *equivalent* to each other. Formally, we define *record stream equivalence* based on multi-set equality [8] as follows:

Definition 4.2 (Record Stream Equivalence). Two record streams $S[\mathbb{S}]$ and $S'[\mathbb{S}]$ are *equivalent*, denoted by $S[\mathbb{S}] \equiv S'[\mathbb{S}]$, iff their corresponding multi-sets of timestamp-value pairs are equal:

$$S \equiv S' \Leftrightarrow \texttt{MS}(\pi_{T,V}(S)) = \texttt{MS}(\pi_{T,V}(S')) \quad (1)$$

with `MS` being an operator that transforms a sequence into a multi-set and π a stream projection that returns the timestamp and value attribute for each record.

4.1 Stateless Operators

Stateless operators as discussed in Section 2.2, among them `filter`, `projection`, `map`, and `flatMap` can be realized for record streams in our model in a straightforward manner. Those operators are applied to each record of the input stream in offset order and produce an (order preserving) output record stream. The output record(s) inherit the timestamp from the input record. Based on record stream equivalence, we define correctness for stateless operators on unordered input streams as follows:

Definition 4.3 (Operator Correctness). Given an ordered data streams $S[\mathbb{S}]$ and an unordered data streams $S'[\mathbb{S}]$. An stateless stream processing operators O is correct, iff:

$$S \equiv S' \Rightarrow O(S) \equiv O(S') \quad (2)$$

Definition 4.3 states that if two input streams are equivalent, both output streams must belong to the same equivalence class. Furthermore, ordered streams are canonical representatives for each equivalence class. Hence, defining operator semantics based on ordered input streams that yield ordered output streams is sufficient to define operator semantics for unordered input and output streams implicitly.

4.2 Record Stream Aggregation

Following our Dual Streaming Model as introduced in Section 3, a record stream aggregation operator `agg` yields an ever updating result table. To define correctness of the `agg` operator for unordered input streams, we use table equality: Given two equivalent input record streams we require `agg` to return the same result table for both.

Definition 4.4 (Operator Correctness). Given an ordered data stream $S[\mathbb{S}]$ and an unordered data stream $S'[\mathbb{S}]$. The `agg` operator is correct, iff:

$$S \equiv S' \Rightarrow \texttt{agg}(S) = \texttt{agg}(S') \quad (3)$$

Note that table equality implies that both tables must contain the exact same table versions. Furthermore, materializing the output changelog stream must yield the original result table. This implies, that table changelog records get assigned the maximum timestamp over all their input records. The required equality of result tables dictates that the `agg` operator updates its state correctly and emits corresponding update records if out-of-order data is processed. We illustrate the correct behavior with an example.

Example 4.5 (Aggregation with out-of-order records). Consider a summation operator that is evaluated over a record stream comprising three records. The records are defined in terms of timestamp-value pairs (omitting the keys, which are set as $k = \perp$ for all records) as follows:

$$\langle 3, 2.3\rangle, \langle 7, 4.4\rangle, \langle 5, 6.1\rangle$$

The correct aggregation result is defined over the corresponding ordered input stream as a table T with three versions T_3, T_5, T_7. This table evolves over three generations as follows (we only show the sum):

$$\begin{aligned}
\langle 3, 2.3\rangle \rightarrow \quad & T(1): \quad T_3 : 2.3 \\
\langle 5, 6.1\rangle \rightarrow \quad & T(2): \quad T_3 : 2.3 \quad T_5 : 8.4 \\
\langle 7, 4.4\rangle \rightarrow \quad & T(3): \quad T_3 : 2.3 \quad T_5 : 8.4 \quad T_7 : 12.8
\end{aligned}$$

The corresponding output table changelog stream consists of three records $\langle 3, 2.3\rangle, \langle 5, 8.4\rangle, \langle 7, 12.8\rangle$.

Processing the input stream with out-of-order records evolves the output table differently:

$$\begin{aligned}
\langle 3, 2.3\rangle \rightarrow \quad & \hat{T}(1): \quad \hat{T}_3 : 2.3 \\
\langle 7, 4.4\rangle \rightarrow \quad & \hat{T}(2): \quad \hat{T}_3 : 2.3 \quad \hat{T}_7 : 6.7 \\
\langle 5, 6.1\rangle \rightarrow \quad & \hat{T}(3): \quad \hat{T}_3 : 2.3 \quad \hat{T}_5 : 8.4 \quad \hat{T}_7 : 12.8
\end{aligned}$$

The respective table changelog stream comprises four records:

$$\langle 3, 2.3\rangle, \langle 7, 6.7\rangle, \langle 5, 8.4\rangle, \langle 7, 12.8\rangle$$

Note that table version $\hat{T}_7 \in \hat{T}(2)$ and the corresponding changelog record $\langle 7, 6.7\rangle$ are *correct*. The stream prefix of length 2 of the ordered and unordered input record stream are not equivalent and thus, both output tables $T(2)$ and $\hat{T}(2)$ do not need to be equal after the first two records have been processed (Definition 4.4) for each case [8].

Result Completeness vs. Runtime Cost: The result table of an aggregation as defined above grows unbounded because the complete history of all table versions is maintained. Hence, in practice it is required to bound the size of the result table [8]. To this end, we consider a *retention-time* parameter, that defines for how long an element in the result table should be maintained. While event-time progresses, elements older than "current event-time minus retention-time" can be deleted from the table. Purging data from the result table implies that further updates to those elements are not accepted and corresponding input records would be dropped.

Introducing a retention-time parameter makes the trade-off between result completeness and storage requirements explicit. For example, if users know that there is an application specific upper bound for late arriving data, they can configure the retention-time accordingly to ensure a complete result and estimate the memory/storage requirements. As such, the retention-time parameter is somewhat similar to a punctuation/watermark mechanism [3, 27]. However, retention-time does not introduce additional latency, because the result table is updated immediately, while a punctuation/watermark would delay the computation until the watermark passes.

Grouping and Windowing Aggregations in stream processing are usually based on a grouping attribute (similar to a group-by operator in the relational model) and time windows. Both concepts are supported in our model by encoding the grouping attribute and a window ID [20] in the result table's primary key. The table stores the current aggregate per key and per window.

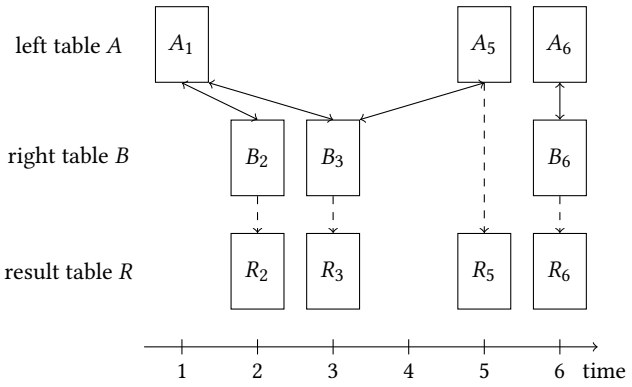

Figure 5: Table-table inner join example.

4.3 Table Operators

So far, we discussed that tables can be materialized from a table changelog stream or can be the result of an aggregation operation. For the former, we introduce a `table` operator that materializes a table changelog stream. Additionally, our model enables a straight-forward realization of `filter`, limited[2] forms of `projection` and `map`, of `agg`, and primary key equi `join` operators for tables. All those operators yield a result table that is updated each time the input table is updated. The semantics of the above table operators follow a temporal-relational model: Updates on an input table version t trigger updates on output table version t. Furthermore, table-table joins are defined over the corresponding table version, i. e., the resulted table with version t should be generated from the left table version t joining with the right table version t.

Example 4.6 (Table-table join). Figure 5 shows an inner join example for input table A (with versions 1,5,6) and input table B (with versions 2,3,6). Note that if there is no update for a certain time t for a table, we can safely use the older table version (i. e., $t-1$) in the join since $T_{t-1} = T_t$. In this example, there is no result table R_1 because there is no table B_1 and thus A_1 cannot be joined at timestamp 1. Because A_2 does not exist, B_2 joins with A_1 yielding R_2. Updating B_2 to B_3 results in updating R_2 to R_3. Similarly, R_5 is the result of joining updated A_5 with B_3. Note that at timestamp 6, both tables are updated at the same time and thus $R_6 = A_6 \bowtie B_6$.

We can describe the result tables as materialized views and use techniques known from relational database systems to maintain the result tables when the input tables are updated [9, 14]. In practice, not all tables need to be materialized

[2] `projection` and `map` are not permitted to alter the table's primary key.

and a cost model may be employed to decide, if an operator uses a materialized table or operates over the corresponding table changelog stream only.

4.4 Stream Based Joins

We also include a variety of equi-join operators for streams in our model, i.e., both stream-stream and stream-table joins.

Stream-stream joins are based on join attributes as well as a sliding window that effectively defines an event-time band-join over both streams: Two records join, if they have the same join attribute and if their timestamps are close to each other, i. e., their timestamp difference is less than or equal to the window size. Join result records get the larger timestamp of both input records assigned.

Stream-table joins perform a primary key table lookup-join and yield an output stream. For each stream record the join attribute is used to look for an equal primary key in the table—if the key is found in the table, the stream record joins with the table record and the join record is appended to the output stream. A stream-table join is a *temporal* join: The table lookup is done into the table version corresponding to the stream record's timestamp. Thus, a stream record can only join with table records with a timestamp less than or equal to the record's timestamp. The join result record inherits the timestamp for the input stream record.

Example 4.7 (Stream-table join). Figure 6 shows a stream-table join with table $T(5)$ from Example 3.1. Stream input records are denoted as timestamp-key-value triples, e. g., $\langle 2, B, 3.5\rangle$ refers to a record with timestamp 2, a join key B and a value 3.5. The first stream record $\langle 2, B, 3.5\rangle$ does not join, because at its timestamp $t = 2$ there is no table available. Record $\langle 5, A, 4.2\rangle$ joins successfully with table version 5 producing output record $\langle 5, A, 4.2 \bowtie 7.2\rangle$. For the third record $\langle 6, C, 6.4\rangle$ there is no corresponding element with $k = C$ in the table and thus no join result is emitted. Finally, record $\langle 7, B, 1.2\rangle$ results in join record $\langle 7, B, 1.2 \bowtie 14.7\rangle$. Note, that the last result record is a join result with version 6 of the table.

A stream-table join can also be a left-outer-join ensuring that there is exactly one join result per stream input record. For the stream-stream join, inner-, left-outer, and full-outer-join semantics can be realized directly in our model.

Out-of-order records: Handling out-of-order records in joins requires several strategies. For stream-table joins, out-of-order records do not require special handling. However, out-of-order table updates could yield incorrect join results, if not treated properly. Assume that the table update in Figure 6 from $\langle A, a, 2\rangle$ to $\langle A, a', 5\rangle$ is delayed. Stream record $\langle A, \alpha', 6\rangle$ would join with the first table version and incorrectly emit $\langle A, \alpha' \bowtie a, 6\rangle$. To handle this case, it is required to buffer

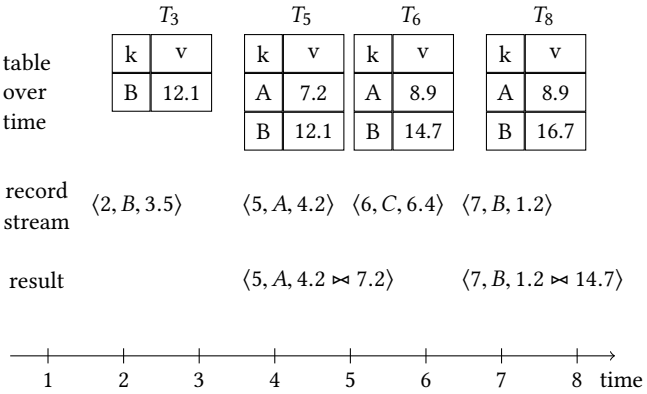

Figure 6: Stream-table join example.

record stream input record in the stream-table join operator and re-trigger the join computation for late table updates. Thus, if a late table update occurs, corresponding update records are sent downstream to "overwrite" previously emitted join records. Note, that the result of stream-table joins is not a record stream but a regular data stream because it might contain update records.

For left- and full-outer stream-stream joins the same technique applies. Records with no join partner are emitted as left-/full-outer join result eagerly and might be updated if a join record appears in the other stream later on. This technique allows to implement the joins in a fully non-blocking manner. For inner stream-stream joins, the output stream is still a record stream though, because the inner stream-stream join is a monotonic operation and can never yield an early result that needs to be updated later on.

Both joins are subject to a retention-time parameter as introduced for tables. If a late stream record is processed after the corresponding table version or stream record was purged, no join result would be produced for inner-join.

5 CASE STUDY: APACHE KAFKA

In this section, we illustrate how to leverage the Dual Streaming Model to design and implement a stream processing system in practice. We use *Apache Kafka* [16], a large-scale streaming platform that is widely adopted in the industry as a concrete example in our study. Kafka stores data streams in *topics* that clients produce to and consume from. Each topic is maintained as a partitioned log where immutable sequence of records can be continually appended by producer clients. Consumer clients can subscribe to one or more topics and read the records from the logs in append order per partition. Records in the log are stored as *key-value* pairs. Each record has an embedded *timestamp* field that is set by the producer clients when they are generated. In addition, each record will be assigned with an *offset* when appended to the log that uniquely identifies its position in the log.

Since 2015, we have worked with the Apache Kafka community to implement our model in Kafka Streams (first released in Apache Kafka 0.10), a client library that enables developers to build their real-time processing applications with data streams stored in Kafka. Today, the Kafka Streams library has been widely used in various industries, including The New York Times, Pinterest, LINE, Trivago, etc.[3] The library contains a high-level DSL for developers to specify their application's processing logic, starting from reading the data streams from Kafka topics, transforming input streams into new streams, generating evolving tables, and eventually piping the result data streams back to Kafka topics. In the Kafka Streams DSL, there are two first-class abstractions: a ***KStream*** and a ***KTable***. A *KStream* is an abstraction of a record stream, while a *KTable* is an abstraction of both a table changelog stream and its corresponding materialized tables in the Dual Streaming Model. In addition, users of the DSL can query a KTable's materialized state in real-time.

5.1 Kafka Streams DSL Operators

Table 1 summarizes the available operators along with their input and output types in Kafka Streams DSL. The operator definitions are well aligned with the Dual Streaming Model. For any operator that results in a KTable, users of the DSL can specify whether or not to materialize the resulting table. For those materialized KTables, Kafka Streams provide a set of APIs to allow developers to query their evolving state at real-time. In addition, Kafka Streams maintains the changelog stream of a KTable in a replicated internal Kafka topic. The replicated table changelog stream in Kafka is highly available and thus can be treated as the source of truth for that KTable's evolving state. For example, the KTable's state can always be re-materialized from this changelog stream topic during scaling operations and fault tolerance events, reflecting the duality of streams and tables in the Dual Streaming Model.

To achieve data parallelism, Kafka Streams DSL requires `join` and `agg` operations to always be based on some keys such that operations of different keys can be executed independently. In particular, the DSL introduces a `groupBy` operator for `agg`, similar to the group-by operation in relational databases that takes a record's value as input and returns a grouping attribute. To execute this operator, the input stream is firstly shuffled via an intermediate partitioned Kafka topic, and then each reshuffled stream partition is aggregated based on the grouping attribute in parallel. None of the operator implementations in Kafka Streams is blocking: Whenever a record is received from the source Kafka topics, it will be processed immediately by traversing through all the connected operators specified in the Kafka Streams DSL until it has been materialized to some result KTable, or written back to a sink Kafka topic. During the processing, the record's timestamp will be maintained/updated according to each operator's semantics as defined in Section 4.

[3] https://kafka.apache.org/documentation/streams/

Table 1: Operators with their input and output types

Operator	1st Input	2nd Input	Output
`filter`, `mapValue`	KStream		KStream
	KTable		KTable
`map`, `flatMap`	KStream		KStream
`groupBy` → `agg`	KStream		KTable
	KTable		KTable
`groupBy` + `windowBy` → `agg`	KStream		KTable
`inner-/left-/outer-join`	KStream	KStream	KStream
`inner-/left-/outer-join`	KTable	KTable	KTable
`inner-/left-join`	KStream	KTable	KStream

5.2 Windowing for Stream Aggregations

Kafka Streams provides several knobs to enable developers make trade-off decisions between result completeness, runtime cost, and latency through the reasoning of *time*. In this section, we give one concrete example of these tuning knobs: the windowing operations for stream aggregations.

For KStream `agg` operators, besides the `groupBy`, users can provide another grouping conditions via the `windowBy` operator. The result of this aggregation then becomes a windowed KTable. For example, users can generate a histogram that counts user clicks per hour, where the hourly specification is expressed in the `windowBy` operator and the resulted KTable effectively contains one materialized state per grouped one-hour window. The window can be either a *time window* or a *session window* [4]. Time windows can either overlap with each other or not (i. e., hopping[4] or tumbling windows). Session windows do not overlap, and a session window ends when it does not receive elements for a certain time.

Figure 7 demonstrates an example of a windowed stream aggregation. In line 1, we first create a KStream from a Kafka topic named "click-topic". Then in line 3, we create a windowed KTable that counts the number of clicks per region every 5 minutes. The aggregation key is specified in `groupBy` in line 4, and the windowing clause is specified in line 5. Inside the `windowBy` operator, we define the window length to be 5 minutes, and each window will be advanced by 1 minute (i. e., it is a hopping window). Additionally, we define that each window will be retained for one hour so that any out-of-order data that arrives within an hour after the window has elapsed will still be processed; i. e., the corresponding window's aggregate results will be updated and a new update record is appended to the table changelog stream accordingly.

[4] *Hopping windows* are often called sliding windows. We use the term *sliding window* for join windows that have different semantics (c. f. Section 4.4).

```
KStream<String, Clicks> clickStream = builder.stream("click-topic");

KTable<Windowed<String>,Long> windowedCount = clickStream
    .groupBy((key, clicks) -> clicks.serverRegion.toLowerCase)
    .windowedBy(TimeWindows.of(5*60*1000).advanceBy(1*60*1000).until(60*60*1000))
    .count();
```

Figure 7: Windowed aggregations example in Kafka Streams.

This windowing operation can be leveraged to determine when each window of the resulted table can be considered "complete": After the retention time passed, a window is not updated any longer and the result is *final* and no more data will be accepted to update the window. With longer window retention period, out-of-order records with older timestamps can be processed. Therefore, by customizing a larger retention period, the application is more resilient to out-of-order data. The cost for longer retention times is an increased storage footprint and a longer period before each window's result can be considered complete.

6 RELATED WORK

Our model builds upon existing work on continuous query processing, materialized view maintenance, and relational query languages for stream processing.

Early work on continuous queries was presented by Terry et al. [26] and Lui et al. [22, 23]. They use continuous queries to refine the result if the input data is updated. As such, the respective model is similar to incremental maintenance of materialized view [9]. Our model takes up these ideas on incremental updates and incorporates them for tables and data streams, i. e., for both, static and dynamic views on data.

The chronicle data model [14] builds on chronicles and relations. A chronicle is an unbounded sequence of transaction records that are applied to a relation. The latter is represented as a versioned table, which is similar to our notion of table versions. However, unlike our Dual Streaming Model, the chronical model neglects the temporal dimension in terms of an explicit time domain. Hence, out-of-order arrival of data cannot be captured in the chronical model.

The seminal work on the Continuous Query Language (CQL) [5] exploits the relational model to provide strong query semantics over data streams. It models streams as tables and, hence, is the closest model to our work. However, CQL does not handle out-of-order records, assuming that such issues may be addressed on the technical layer by means of buffering and reordering. Our notion of versioned tables can be seen as generalization of CQL: Ordered streams can be handled as a special case, which allows to purge older table versions immediately upon the creation of a newer version. Trill [10] supports a relational-temporal stream processing model similar to our work. However, as in case of CQL, there is no notion of out-or-order data.

Law et al. [18] suggested to define operators that continuously update the output, thereby inducing a notion of "the result so far". They define correctness of operators base on input prefix and have a formal notion of blocking and non-blocking operators. The difference to our work, however, is that their model targets only record streams. As a consequence, their model is limited to monotonic queries. Furthermore, their model does not consider out-of-order records or window-based operators.

Data streams containing update or delete records have been discussed by Babu and Widom [7] While their work puts forward the general idea, it does not provide semantics of streaming operators and notions of operator correctness. The Borealis system [2] stores the history of an input stream and allows for in-place updates, if late update records arrive. For this case, the stored input stream is replayed after it was updated. We argue that this is a rather expensive approach to handle updates, compromising processing latency. In contrast, our model realizes incrementally updates, which limits changes to the results that actually need to be updated.

The ability for reasoning about time is essential for dealing with unbounded, unordered data streams of varying time skew. The notion of event-time was first introduced by Srivastava and Widom [25]. This work points to the related synchronization issues and suggests buffering and reordering techniques to handle out-of-order data. Moreover, Barga et al. [8]. introduced strong time semantics for streams and, similar to temporal database systems, and propose definitions for different levels of consistency. The SPREAD model [15] compares the timestamps and arrival order of records to identify out-of-order data. Here, the key insight is that evaluation triggered by the arrival of a batch of records (e. g., scoped by the advance of the tuples' timestamps) cannot comprehensively capture the temporal aspects of the incoming data. The solution proposed in SPREAD is to define batches of records with the same timestamp, i. e., evaluations can be triggered multiple times within the same timestamp, following the arrival order.

Finally, out-of-order data may be handled by *punctuations* [27]. They represent control messages that provide certain guarantees about the data stream. Watermarks [3, 4] are a special form of punctuation that report lower bounds for event-time progress. The discussed techniques have in common that they imply partially blocking until a watermark or

punctuation arrives. As highlighted by Maier et al. [24], such an approach inherently increases the processing latency.

Li et al. [21] and Krishnamurthy et al. [17] realized that in-order processing has disadvantages like increased latency and memory requirements. They suggest out-of-order processing similar to our model, yet, do not include a notion of a changelog stream. Thus, they rely on punctuations to delay computation [21] and query results are not updated continuously [17].

7 CONCLUSION

In this paper, we introduced the Dual Streaming Model. At its core is the idea to specify the result of a stream processing operator as successive updates to a table. These updates may either be materialized into a versioned table or may be represented as a stream of update records, which induces a duality of streams and tables. Our Dual Streaming Model enables us to reason about inconsistencies in the physical and logical order of data in a stream in an holistic manner: it provides well-defined operator semantics for processing unordered streams continuously, without explicit buffering and reordering of records. This way, the latency of stream processing is not compromised.

To handle of out-of-order data directly in the stream processing model has further advantages. We discussed how our model makes explicit the trade-offs between result completeness, processing cost, and latency in data stream processing environment. Finally, we presented an implementation of the Dual Streaming Model in Apache Kafka, a widely adopted open-source stream processing platform.

8 ACKNOWLEDGMENTS

We thank Michael Noll, Damian Guy, Hojjat Jafarpour, Bill Bejeck, Sriram Subramanian, and Jay Kreps for extensive discussions that lead to this paper, and their work on Kafka Streams.

REFERENCES

[1] Daniel Abadi et al. 2003. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal* 12, 2 (2003), 120–139.

[2] Daniel Abadi et al. 2005. The Design of the Borealis Stream Processing Engine. In *CIDR, 2nd Biennial Conf. on Innovative Data Systems Research*. 277–289.

[3] Tyler Akidau et al. 2013. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.* 6, 11 (2013), 1033–1044.

[4] Tyler Akidau et al. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *Proc. VLDB Endow.* 8, 12 (2015), 1792–1803.

[5] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2003. CQL: A Language for Continuous Queries over Streams and Relations. In *Database Programming Languages, 9th Int. WS*. 1–19.

[6] Brian Babcock et al. 2002. Models and Issues in Data Stream Systems. In *Proc. of the 21st ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*. 1–16.

[7] Shivnath Babu and Jennifer Widom. 2001. Continuous Queries over Data Streams. *SIGMOD Records* 30, 3 (2001), 109–120.

[8] Roger Barga et al. 2007. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *CIDR, 3rd Biennial Conf. on Innovative Data Systems Research*. 363–374.

[9] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. 1986. Efficiently Updating Materialized Views. *SIGMOD Record* 15, 2 (1986), 61–71.

[10] Badrish Chandramouli et al. 2014. Trill: A High-performance Incremental Query Processor for Diverse Analytics. *Proc. VLDB Endow.* 8, 4 (2014), 401–412.

[11] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.* 44, 3 (2012), 15:1–15:62.

[12] Nihal Dindar et al. 2013. Modeling the Execution Semantics of Stream Processing Engines with SECRET. *The VLDB Journal* 22, 4 (Aug. 2013), 421–446.

[13] Jim Gray et al. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery* 1, 1 (1997), 29–53.

[14] H. V. Jagadish, Inderpal Singh Mumick, and Abraham Silberschatz. 1995. View Maintenance Issues for the Chronicle Data Model (Extended Abstract). In *Proc. of the 14th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*. 113–124.

[15] Namit Jain et al. 2008. Towards a Streaming SQL Standard. *Proc. VLDB Endow.* 1, 2 (2008), 1379–1390.

[16] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*. 1–7.

[17] Sailesh Krishnamurthy et al. 2010. Continuous Analytics over Discontinuous Streams. In *Proc. of the 2010 ACM SIGMOD Int. Conf. on Management of Data*. 1081–1092.

[18] Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. 2004. Query Languages and Data Models for Database Sequences and Data Streams. In *Proc. of the 13th Int. Conf. on Very Large Data Bases*. 492–503.

[19] James Lewis and Martin Fowler. 2014. Microservices: a definition of this new architectural term. https://www.martinfowler.com/articles/microservices.html

[20] Jin Li et al. 2005. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*. 311–322.

[21] Jin Li et al. 2008. Out-of-order Processing: A New Architecture for High-performance Stream Systems. *Proc. VLDB Endow.* 1, 1 (2008), 274–288.

[22] Ling Liu, Calton Pu, and Wei Tang. 1999. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge Data Engineering* 11, 4 (1999), 610–628.

[23] Ling Liu et al. 1996. Differential Evaluation of Continual Queries. In *Proc. of the 16th Int. Conf. on Distributed Computing Systems*. 458–465.

[24] David Maier et al. 2005. Semantics of Data Streams and Operators. In *Proc. of the 10th Int. Conf. on Database Theory*. 37–52.

[25] Utkarsh Srivastava and Jennifer Widom. 2004. Flexible Time Management in Data Stream Systems. In *Proc. of the 23rd ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*. 263–274.

[26] Douglas Terry et al. 1992. Continuous Queries over Append-only Databases. In *Proc. of the 1992 ACM SIGMOD Int. Conf. on Management of Data*. 321–330.

[27] Peter Tucker et al. 2003. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Transactions on Knowledge Data Engineering* 15, 3 (2003), 555–568.