XStreamify: Vision for an Efficient Embeddable Streaming System

Jannik Wolff[1,†]

[1]TU Dortmund University, Germany

Abstract

This paper introduces XStreamify and the vision for an efficient embeddable streaming system designed to overcome some limitations of existing distributed and resource-heavy streaming solutions. Inspired by the success of embeddable data processing systems like SQLite, XStreamify aims to provide stream processing capabilities that are tightly integrated into applications. We outline key requirements for XStreamify, including high efficiency even in resource-constrained environments and seamless, safe embedding within applications. We then detail our research goals, addressing how the embedded nature allows for unique performance optimizations and tight application integration. XStreamify aims to provide lightweight and high-performance stream processing while balancing the flexibility of custom code with the robustness of a general streaming system.

Keywords

Stream Processing, Streaming Systems, Embedded Data Processing

1. Introduction

Stream processing is a paradigm for processing incoming data in real-time. It is particularly useful when results are expected with low latency. They are in many ways similar to traditional database systems, but are designed to handle unbounded data sets.

Most established streaming systems are designed to be used in a distributed environment. Thus, their initial setup is often complex and requires planning and commitment to a specific architecture (e.g. Apache Flink requires cluster environment like Kubernetes to run). It is therefore difficult to use existing streaming systems in small applications or applications that did not initially plan for stream processing.

In other areas of data processing, embeddable systems have proven to be useful in similar situations. SQLite allows performing OLTP workloads in applications without the need for a separate database server [1]. Integration happens in the form of a library and is widely available for many programming languages and runtime environments. More recently, DuckDB has emerged as an embeddable system for OLAP workloads [2]. Similar to SQLite, it aims to be portable and easy to integrate into applications.

There is no clear option for a streaming system that can be embedded into an application in the same way as SQLite or DuckDB. We propose XStreamify, a streaming system that can be embedded into applications. We identified the following requirements for our system:

- High efficient execution of streaming workloads, especially since our target application may be resource constrained.
- While we cannot provide full fault tolerance as an embeddable system, we still need to provide some guarantees. Our system should be able to recover from failures and limit the impact of missing data. Ultimately, our offering must be competitive with existing streaming systems.
- Easy and safe embedding of the system. For one, this requires a portable design as to not have many requirements for the embedding application. Secondly, we must take special care as the

36th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), September 29 - October 01, 2025, Regensburg, Germany

† Supervised by Prof. Dr. Jens Teubner

EMAIL: jannik.wolff@tu-dortmund.de (J. Wolff)
ORCID: 0009-0005-7096-7596 (J. Wolff)

system runs in the same process as the application, as such errors and faults must be contained and not crash the application.

With our work on XStreamify, we aim to provide results on the following research questions:

1. How can we take advantage of being embedded into the application? This position allows for unique performance optimizations and efficient data transfer that standalone systems cannot provide. With standalone systems, data transfer is often done over the network and requires serialization.

2. How can we apply techniques such as JIT compilation to stream processing? JIT compilation has been thoroughly explored for database systems [3, 4] and initial research has also been done for streaming systems [5]. Since we envision XStreamify to be efficient and achieve performance on par with existing streaming systems, JIT compilation is an attractive option.

3. How can streaming pipelines be optimized adaptively? Besides physical optimization, we may also apply logical optimizations that apply to the processing pipeline. Since data streams are unbounded, queries may run for a long time, and initial optimization decisions may not be effective anymore as data distribution changes. We therefore need ways to detect such situation and next, require ways to adapt the active processing pipeline.

4. How can tight integration into the consumer application be achieved? We want to provide an elegant and user-friendly way to specify processing pipelines. This includes allowing users to define their own operators, which is a common feature in existing streaming systems. We also want to allow users to integrate sources and sinks defined by the application. This question is closely tied with the first question.

5. How can XStreamify be transparent and understandable to the user? Ideally, users should not treat XStreamify as a black box, but instead as an open part of their application. This may entail providing introspection capabilities and debugging tools.

In this paper, we begin by discussing the context of work. We briefly describe streaming systems and their key characteristics. Next, we look at some existing embeddable data processing systems and identify strengths and weaknesses. Then we go into detail about our system design goals for XStreamify and how we plan to achieve them.

2. Streaming Systems

Streaming systems are designed to process data in real-time as it arrives, rather than storing it for later processing. Commonly, stream pipelines are composed of sources, operators, and sinks.

Sources connect producers of data to the streaming system. Those can be external systems, such as message queues or databases. The application itself may also act as a data source, allowing for much flexibility when it comes to data ingestion.

Next, operators process the data. Basic stateless operators are filtering and mapping. More complex operations are aggregations and joins. Since data is unbounded, these operations are often performed on windows of data. Thus, they are inherently stateful. This complicates their implementation as they are harder to parallelize, while providing correct semantics.

For extensibility, user-defined operators are often supported. This can either mean the streaming system calls into the user-defined code, or the user-defined code is packaged and executed in the context of the streaming system.

Finally, sinks connect the streaming system to consumers of data. Again, sinks can be external systems (e.g. databases) or the application itself.

Figure 1 shows the components of a streaming system and the application boundary.

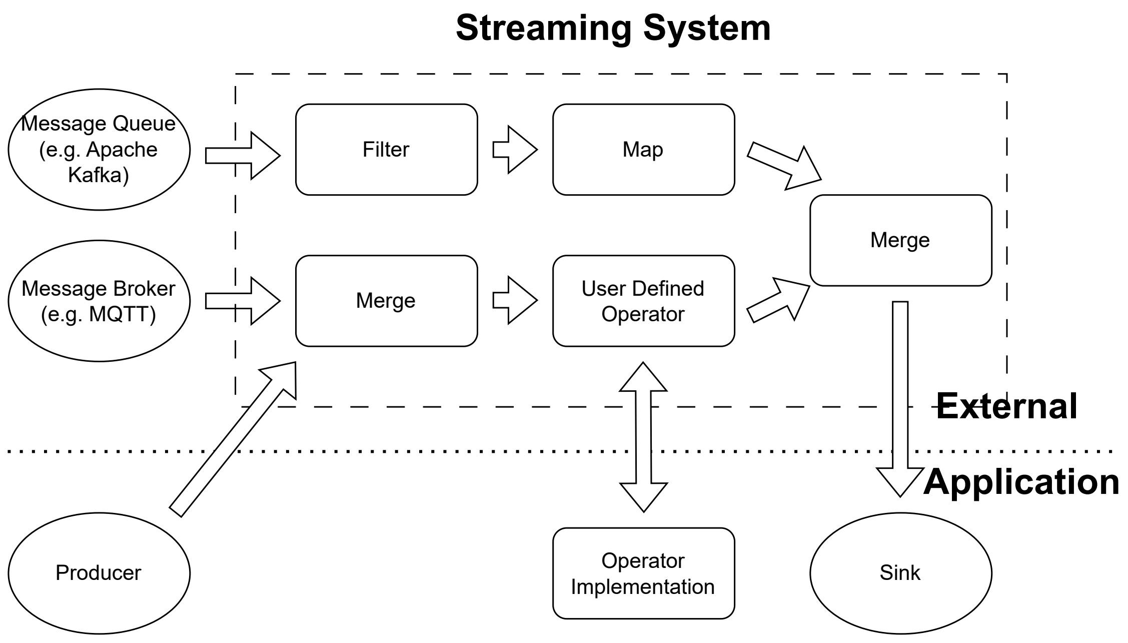

Figure 1: Components of a streaming system with the application boundary.

3. Embeddable Data Processing Systems

XStreamify will compete in the space of embeddable data processing systems. We therefore look at some representative systems in this space and their characteristics. On the one hand, this allows us to take inspiration from their design and implementation. On the other hand, we can identify their limitations and potentially address them in XStreamify.

The most prominent examples of embeddable data processing systems are SQLite [1] and DuckDB [2]. They are designed for OLTP and OLAP workloads, respectively. Their key characteristics are portability (being implemented in C and C++, respectively) and ease of integration. This makes them usable in a wide range of applications and programming languages. SQLite especially stands out for its extensive test coverage and the resulting stability.

There also exist embeddable streaming systems. Noticeably, these systems do not achieve the same level of portability as SQLite or DuckDB. As such, Apache Samza [6] has an operating mode suitable for embedding into applications, while Hazelcast Jet [7] is specifically designed to be embedded. However, both systems are JVM based and therefore require a relatively heavy runtime environment. This may be problematic when using non-JVM languages or when resource constraints are a concern. Similarly, Quix Streams [8] and Bytewax [9] are Python libraries for stream processing. Thus, they are limited to Python applications. Quix Streams implements all processing in Python, which may lead to performance issues in high throughput scenarios. Bytewax, on the other hand, shifts core processing to Rust, which allows for better performance and provides a Python interface.

As embeddable systems, they cannot provide the same level of fault tolerance as standalone systems, which may be distributed inside a cluster. There are different approaches to address this. For example, Quix Streams pushes some state to Apache Kafka, which allows for recovery in case of failures. This in turn, of course requires a running Kafka cluster, thus losing being lightweight.

4. System Design Goals

When considering the requirements and research questions, we identified the following design goals for XStreamify. We explain each goal in more detail in the following sections and state our approach to achieve them.

4.1. Application Integration

Streaming applications are coupled with the application they are embedded into. Users must be able to input data into the streaming system and receive results.

A key question is how stream pipelines are specified. Pipelines consist of sources, operators and sinks. Users should be able to define pipelines in a declarative, yet expressive and user-friendly way. Similar to SQL queries, declarative specifications allow users to focus on the data flow and processing logic, while our system can optimize the execution plan.

Ideally, we do not limit XStreamify to just a single specification method and develop an intermediate representation (IR) that allows expressing such specifications. This IR must be able to encode and model different stream processing semantics [10, 11]. Thus, we can support existing languages (e.g. CQL [12]), DataFrame APIs [13] in an application language or a custom domain specific language (DSL). Figure 2 shows how the same specification may be expressed in three different forms. We plan on implementing a DSL learning from the success and shortcomings of SQL [14, 15] and other query languages. We are particularly interested in a DSL that provides abstractions in order to allow users to reuse components and structure their streaming specifications. The example in figure 2 shows a simple filter operation being abstracted into a reusable function. Of course, this becomes more useful with more complex operations.

```
// CQL
SELECT sensor_id, AVG(temperature) AS avg_temp
FROM SensorStream
WHERE country = 'Germany'
GROUP BY sensor_id
KEEPING 10 SECONDS;

// Hypothetical DataFrame API
sensor_stream
  .filter(s: s.country == "Germany")
  .groupBy(s: s.sensor_id, window("10 seconds"))
  .agg(avg_temp = avg("temperature"));

// Hypothetical DSL
let my_filter = stream:
  stream |> filter (s: s.country = 'Germany')
in
source("sensor_stream")
|> my_filter
|> sliding_window (time.seconds 10)
|> group_by (s: s.sensor_id) (s: {
  s.sensor_id;
  avg_temp = avg s.temperature;
})
```

Figure 2: Example for a semantically equivalent stream processing specification expressed in CQL, a DataFrame-API and a DSL.

It is common for streaming systems to allow users to define their own operators [16]. This allows for application specific processing logic that is not covered by the built-in operators. We want to allow users to define such operators in the programming language of the host application. This comes with the challenge of passing data between the streaming system and the user-defined operators. Here, we see large potential in being embedded into the application. Since we share the same address space, there is potential for very efficient data transfer to user-defined operators.

4.2. Physical Optimization

As mentioned before, we aim for XStreamify to be efficient. We plan on implementing JIT compilation techniques to optimize the execution of streaming pipelines. This is a much more favorable approach than interpreted execution, as it allows for higher throughput and lower latency [5]. The obvious drawbacks are increased complexity and system architecture dependency with code generation and compilation. In this area, we can build on previous work of the DBIS group at TU Dortmund University [4, 17].

When it comes to the interaction with the host application, we will look into efficient data transfer mechanisms. Being embedded into the application and sharing the same address space, allows us to avoid serialization and network overhead. Using data transfer formats such as Apache Arrow [18] may allow for even faster data transfer, using zero-copy transfers and cache-friendly data representation.

4.3. Logical Optimization

The optimizations for executing the physical plan are only one part of the optimization process. Their effectiveness highly depends on the logical plan that they are applied to. Database query plan optimization techniques are well established, but they rely on cardinality estimates and data distribution statistics. In streaming systems, data is often unbounded and the distribution of data may change over time. Therefore, the first challenge is developing an efficient execution plan. Secondly, as the processing may be long-running, and the data distribution may change, we want to be able to adapt the execution plan. For this, we first need to be able to monitor the execution of the processing pipeline. This can be done though instrumentation, measuring throughput and latency of operators, and measuring the data distribution. Additionally, we may even be able to use CPU performance counters to gain some of the required information. All in all, this must be done in a way that does not introduce too much overhead.

Finally, based on the measurement we have to decide if the active processing pipeline needs to be adapted. Picking a new execution plan may involve replacing operators, changing the order of operators, or even replacing the whole processing pipeline. This is a complex task, as we need to ensure that the new plan is still valid and does not break the semantics of the processing pipeline. Especially in the presence of stateful operators, we need to take care of not losing or invalidating state.

4.4. Introspection and Debugging

We aim for integrating XStreamify into an application not to mean adding a black box to the application. Instead, we want to provide users with the ability to inspect the active streams and their operators. Critical parts of an application may move into the streaming system, and it is therefore essential still understand these parts.

On a high level view, users should be able to see active streams and their operators. This includes information about data distribution and throughput across the operators. Debugging tools should allow users to inspect the behavior of operators and their state.

On a lower level, advanced users may want to inspect the internal state of the streaming system. This may include internal data structures, such as the operator state, and performance counters. Additionally, inspecting the execution plan and the JIT compiled code may be of interest. To make this possible, generated code must be augmented with debug information that allows matching the code to the operators of the processing pipeline.

5. Use Cases and Impact

Currently, developers face the decision of how to implement stream processing in their applications. They can either implement stream-like logic in their application code or use an existing standalone streaming system. The first option has the advantage of being very application specific and flexible. However, the initial implementation requires implementing certain primitives, such as buffering and windowing. This can be easy at first, but it quickly becomes hard to maintain and generalize. The second option, using an existing streaming system, requires changing the whole system architecture. This often involves getting familiar with a new system and its concepts. Developers are able to reuse existing components, but the integration into the application is often not as seamless as with the first option.

We aim to provide a novel third option with XStreamify. Balancing the tradeoffs of the first two options, we allow for tight integration into the application while still providing a general streaming system. Integration and onboarding should be easy, while many complex internal details are hidden behind abstractions.

Additionally, we see XStreamify as a potential solution for resource constrained environments, such as edge computing. In such environments, it may not be feasible to run a full-fledged streaming system. Thus, we provide a lightweight solution that can run on resource constrained devices.

6. Conclusion

We introduced XStreamify, a lightweight embeddable streaming system designed to give applications stream processing capabilities. XStreamify will be our vehicle for doing research in the area of stream processing.

We outlined our design goals for the system. First, we aim for highly efficient stream processing, by applying techniques such as JIT compilation to stream processing. Next, we will look at declarative pipeline specifications and representations allowing for logical optimizations. Finally, as an embeddable system, we want to reach especially tight application integration. This includes user-defined operators in application code, introspection of the system, and debugging capabilities.

Our next steps will be implementing a prototype of XStreamify, first adding the foundational components for executing streaming workloads. We plan to keep the implementation compartmentalized as much as possible, allowing us to target our goals step by step. For example, adding a user-facing layer for application integration and stream specification. Then, as a separate layer on top, integrating capabilities for introspection and debugging. On the opposite side, we can then implement an internal execution layer taking platform specific optimizations into account, while being independent of the user-facing layer.

Declaration on Generative AI

During writing, the author used GitHub Copilot as a smart autocompletion tool for single sentences. Most sentences were subsequently altered. The author used Gemini 2.5 Flash to create a draft for the abstract, which was subsequently revised and shortened. Finally, the author used GPT-4o to suggest grammar and spelling improvements, which were manually reviewed and applied. After using these tools, the author reviewed and edited the content as needed and takes full responsibility for the publication's content.

References

[1] K. P. Gaffney, M. Prammer, L. C. Brasfield, D. R. Hipp, D. R. Kennedy, J. M. Patel, SQLite: Past, Present, and Future, Proc. VLDB Endow. 15 (2022) 3535–3547. doi:10.14778/3554821.3554842.

[2] M. Raasveldt, H. Mühleisen, DuckDB: An Embeddable Analytical Database, in: Proceedings of the 2019 International Conference on Management of Data, ACM, Amsterdam Netherlands, 2019, pp. 1981–1984. doi:10.1145/3299869.3320212.

[3] T. Neumann, Efficiently compiling efficient query plans for modern hardware, Proceedings of the VLDB Endowment 4 (2011) 539–550. doi:10.14778/2002938.2002940.

[4] H. Funke, J. Mühlig, J. Teubner, Efficient generation of machine code for query compilers, in: Proceedings of the 16th International Workshop on Data Management on New Hardware, ACM, Portland Oregon, 2020, pp. 1–7. doi:10.1145/3399666.3399925.

[5] P. M. Grulich, B. Sebastian, S. Zeuch, J. Traub, J. V. Bleichert, Z. Chen, T. Rabl, V. Markl, Grizzly: Efficient Stream Processing Through Adaptive Query Compilation, in: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, ACM, Portland OR USA, 2020, pp. 2487–2503. doi:10.1145/3318464.3389739.

[6] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, R. H. Campbell, Samza: Stateful scalable stream processing at LinkedIn, Proceedings of the VLDB Endowment 10 (2017) 1634–1645. doi:10.14778/3137765.3137770.

[7] C. Gencer, M. Topolnik, V. Ďurina, E. Demirci, E. B. Kahveci, A. Gürbüz, O. Lukáš, J. Bartók, G. Gierlach, F. Hartman, U. Yılmaz, M. Doğan, M. Mandouh, M. Fragkoulis, A. Katsifodimos, Hazelcast jet: Low-latency stream processing at the 99.99 $^{\text{th}}$ percentile, Proceedings of the VLDB Endowment 14 (2021) 3110–3121. doi:10.14778/3476311.3476387.

[8] Quix Streams: Python Streaming DataFrames for Kafka, Quix Analytics, 2025.

[9] Bytewax: Python Stream Processing, Bytewax, Inc., 2024.

[10] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, N. Tatbul, SECRET: A model for analysis of the execution semantics of stream processing systems, Proceedings of the VLDB Endowment 3 (2010) 232–243. doi:10.14778/1920841.1920874.

[11] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, S. Whittle, The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing, Proceedings of the VLDB Endowment 8 (2015) 1792–1803. doi:10.14778/2824032.2824076.

[12] A. Arasu, S. Babu, J. Widom, The CQL continuous query language: Semantic foundations and query execution, The VLDB Journal 15 (2006) 121–142. doi:10.1007/s00778-004-0147-z.

[13] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, Apache Spark: A unified engine for big data processing, Communications of the ACM 59 (2016) 56–65. doi:10.1145/2934664.

[14] J. Shute, S. Bales, M. Brown, J. Browne, B. Dolphin, R. Kudtarkar, A. Litvinov, J. Ma, J. D. Morcos, M. Shen, D. Wilhite, X. Wu, L. Yu, SQL has problems. We can fix them: Pipe syntax in SQL, Proc. {VLDB} Endow. 17 (2024) 4051–4063. doi:10.14778/3685800.3685826.

[15] T. Neumann, V. Leis, A Critique of Modern SQL and a Proposal Towards a Simple and Expressive Query Language, in: Proceedings of the 14th Conference on Innovative Data Systems Research, www.cidrdb.org, 2024.

[16] M. Fragkoulis, P. Carbone, V. Kalavri, A. Katsifodimos, A survey on the evolution of stream processing systems, The VLDB Journal 33 (2024) 507–541. doi:10.1007/s00778-023-00819-8.

[17] H. Funke, J. Mühlig, J. Teubner, Low-latency query compilation, The VLDB Journal 31 (2022) 1171–1184. doi:10.1007/s00778-022-00741-5.

[18] Apache Arrow, 2025.