

CHARACTERISTICS OF A GOOD LANGUAGE FOR STREAM PROCESSING

Any language intended to be used for building stream processing applications should exhibit a number of key characteristics and avoid certain pitfalls, which we outline in this article. Our goal is to provide high-level guidance to information technologists who are interested in designing or evaluating languages for complex, large-scale stream processing applications and services.

In the rest of the article, we first discuss these characteristics under five broad desirable features. We then provide a high-level comparison of alternative stream processing languages on the basis of the desirable features.

1. DESIRABLE FEATURES

- **Support for Stream-Oriented, Time-Series Event Processing**

The language should support stream-oriented operations and constructs that simplify expressing complex processing logic on (windows of) moving time-series data, including handling potentially late or incomplete data. It should also be easy to extend the operator set to achieve new domain-specific functionality.

The language must support specialized stream-oriented operators for filtering, merging, correlating, pattern matching and aggregating on “windows” of time-series data, as these idioms occur very frequently in stream processing applications (e.g., computing a 5-minute rolling average or correlating the portfolio stock average with the sector average for the same time period). Windows define the scope of an operation, indicating what subset of the data stream the operator should process at each point in time. The language should integrate flexible windowing constructs to specify various window types (e.g., including time-based and count-based windows). Overall, the language should be powerful enough to express complex monitoring and transformation rules over streaming data.

Moreover, the language operators must provide support for dealing with imperfections in the data streams, caused by delayed or out-of-order data arrivals, both of which frequently happen in real-world applications. For example, the system should not be forced to block on input data that may arrive too late or, worse, not arrive at all.

Finally, the operator set must be extensible, so that developers can easily achieve new, application-specific functionality within the system (e.g., to implement a proprietary analysis algorithm on the streaming data).

- **Support for Uniform, Integrated Management of Streaming and Stored Data**

The language should have the ability to deal uniformly with both streaming and stored data, ideally using the same operators and constructs.

Many applications require access and manipulation of both live streaming and stored historical data within the same application. For example, in on-line data mining applications (e.g., credit card or other transactional fraud detection), identifying whether an activity is unusual requires gathering the usual activity patterns over time and comparing them to the present activity in real time. Another example comes from firms with electronic trading applications, who want to write a trading algorithm and then test it on historical data to see how it would have performed on alternative scenarios. When the algorithm works well on historical data, the customer wants to seamlessly switch it over to a live feed.

To handle these and similar cases, the language should have the ability to convert streaming data to stored data and vice versa, as appropriate. Ideally, the language operators should be agnostic about the type of data they process and, as such, the same (or very similar) syntax should be used to manage both types while necessary conversions between the types are automatically performed as appropriate based on the semantics of the operator. In general, the scope of an individual operator should be either live data, stored data, or both (e.g., a join between stored and live data).

- **Enabling Data Independence**

The language should provide data independence to hide the details of how the data are represented and maintained inside the system and to decouple the data from applications that access and use the data. Data independence allows the internal structures that manage the data to be modified without affecting the applications. Furthermore, it also allows existing applications to be easily modified (or even removed) or new ones added without affecting the others.

The language should facilitate **data independence** by cleanly separating the data (and its management) from the processes that use the data. This principle is at the heart of nearly all modern, complex systems and is best exemplified by database management systems (DBMSs) where the data is exclusively managed by a separate system independent of the applications. This approach is in sharp contrast with the data dependent approach in which the data is represented and managed *inside the code* of various applications. Such an approach is inflexible as it makes the modification and sharing of data across multiple applications very difficult.

Data independence implies two complementary concepts. **Physical data independence** hides the details of how the data is physically represented and managed by providing a high-level, constrained interface to define and manipulate it. This approach enables the system to easily change and optimize its internal data structures and algorithms without requiring any modifications to the existing applications.

Logical data independence, on the other hand, allows each application to have its own customized view of data. For example, in many cases, a trading application would much rather deal with a single logical market feed that is “clean” and “complete”, than deal with multiple raw market feeds, each potentially dirty and incomplete by itself. Such a logical “super feed” can be derived by cleaning the raw feeds and combining them in real time as they stream into the system. This approach can also hide the differences between the schemas of the raw feeds as well as modifications to the schemas, by presenting a unified logical schema to the application. It is, therefore, possible for the underlying data feeds to change without causing a major disruption in the application. In this case, all that is required is to remap the underlying feeds to the logical view.

Languages that support data independence provide a high-level of abstraction, making the complicated concepts associated with stream processing accessible to the user without the need for low level programming or deep knowledge about the underlying physical resources or infrastructure.

- **Support for Complex Application Development**

The language should simplify the development and maintenance of complex applications. To achieve this goal, it should adopt widely-accepted software engineering design practices and goals, ideally using familiar, standardized syntax and semantics. The language should also avoid operations or control logic that make it difficult to statically reason about the run-time behavior of the application.

Today’s business applications are highly complex and dynamic: customers, markets, business rules and regulations are constantly and rapidly increasing in number and sophistication. To remain competitive and compliant, businesses need the flexibility to quickly adapt to changing business and analytic needs in a cost-effective manner.

To meet these requirements, the language should ease the development, maintenance and incremental evolution of complex applications. To deal with complexity, the language should be highly modular: it should incorporate constructs to define and instantiate modules that can be easily and incrementally composed and reused. Modularity disallows direct interactions between random parts of an application, restricting them to happen through well-defined interfaces between modules.

As applications get increasingly larger and sophisticated, reasoning about their run-time behavior becomes quite challenging. A modular design goes a long way towards guaranteeing predictable, deterministic behavior (as it reduces and localizes side effects) but is alone insufficient. In particular, it has long been widely accepted that non-linear control logic (e.g., go-to statements) is a “no-no”, because it yields difficult to trace, unpredictable executions. The language should, thus, avoid such constructs and lend itself to static code analysis that makes it possible to reason about its behavior independent of run-time conditions.

A critical factor in the success of any language is how easily its syntax and semantics can be understood. A “cryptic” language will certainly receive resistance from potential users. On the other hand, a language that is based on standard, familiar notation and semantics will avoid steep adoption curves and instantly become effectively usable by a large user population.

- **Enabling High-Performance Stream Processing**

The operations in the language should lend themselves to very efficient implementation and optimization to meet the high-volume very low-latency processing needs of stream-based applications.

It is crucial that the implementation of the language include a well-optimized execution strategy that can provide very low response times even in the presence of high volume data. Furthermore, the execution should lend itself to easy, transparent distribution across a cluster of workstations or a multi-core machine, as these multiprocessor configurations are being increasingly commonplace due to their favorable price-performance characteristics.

2. LANGUAGES for STREAM PROCESSING

In addition to custom coding, there are at least three software system technologies that can potentially be used to develop complex stream data processing applications and services. These are database management systems, rule engines, and stream processing engines. Here, we briefly present the languages these technologies are built upon.

- **Rule languages** date from the early 1970’s when they were initially proposed by the artificial intelligence (AI) community. The key construct in a rule language is a condition/action pair, usually expressed using “if-then” notation. As data enters the system, they modify the state of the program variables, and, as such, they are matched against the existing rules. When the condition of a rule is matched, the rule is said to “fire”. The corresponding action(s) taken may then produce alerts/outputs to external applications or may simply modify the state of internal variables, which may lead to further rule firings.
- **SQL** is a high-level, declarative language query language that has remained the enduring standard database language over three decades. SQL is based on the relational algebra and includes a powerful set of operators for filtering, merging, correlating, and aggregating data. SQL is explicit about how these primitives interact so that its meaning can be understood independently of run-time conditions.
- **StreamSQL** is a variant of standard SQL and has recently been specifically designed to express processing over continuous streams of time-series data. Stream SQL can be used to perform SQL-style processing on the incoming messages as they fly by, without necessarily storing them as SQL does. StreamSQL extends SQL by adding to it rich windowing constructs and stream-oriented operations.

2.1 High-level comparison with respect to the desirable features

Support for Stream-Oriented, Time-Series Event Processing. SQL was originally designed to operate on unordered stored data sets and thus needs to be extended to deal with potentially unbounded streams of ordered (time-series) data. For example, SQL, as implemented by the DBMS vendors, supports only a rudimentary notion of windowed operations. Rule languages need to be extended in a similar manner so that they can express complex conditions of interest over time. Both languages can support extensibility. StreamSQL provides native support for stream-oriented, time-series processing whereas the others do not; although they can be extended to do so, the required modifications are non trivial.

Support for Uniform, Integrated Management of Streaming and Stored Data. As argued above, both SQL and rule languages need to be extended to express operations on continuous data streams. Furthermore, rule languages also have problems when dealing with stored, persistent data because they rely on local variables for storage. Thus, a rule language is ill-suited for storing and querying large amounts of persistent state. Thus, it needs to be coupled with a SQL paradigm, and switch from local variables to SQL tables, depending on whether it is dealing with stored or streaming data. StreamSQL, on the other hand, has a distinct advantage, as it was derived from SQL and can seamlessly deal with streaming and stored data by simply switching the scope of a command from a live feed to a stored table.

Enabling Data Independence. Both SQL and StreamSQL are high-level, declarative languages that can provide data independence, both at physical and logical levels. This is one of the key reasons behind the widespread adoption and success of SQL engines as part of virtually all enterprise systems. A rule language, on the other hand, typically extends a traditional low-level programming language with condition-action pairs. The data is represented and stored in the language variables and is tightly integrated in the code, making it difficult to modify applications and share data among them.

Support for Complex Application Development. All three languages can incorporate mechanisms for defining processing modules. SQL and StreamSQL define “views”, or named queries, whose results can be directly used by other queries. In both cases, queries are composed and executed in a statically-defined linear order. Furthermore, because the native language operators have well defined semantics, it is easy to reason about the run-time behavior of an individual query. These two properties together facilitate high modularity and understandability even for very complex applications.

A rule language, on the other hand, offers modularity for only rule management. Rule definitions are encapsulated in a single module as opposed to being interspersed in the code, making their addition, deletion, and management easy. However, at the semantic level, rules have a fundamental problem as they allow non-linear control of execution. In particular, rules can fire in an unpredictable manner that is hard or sometimes impossible to statically predict. The possibility of cascaded rule firings aggravates the problem. As

applications become more complex and rule collections get bigger, it becomes increasingly more challenging to decide whether a given rule collection will produce the intended run-time semantics. Moreover, seemingly incremental changes to the rule collection may well yield non-trivial changes to the global behavior, due to the non-linear, potentially complex interactions among the rules. Sometimes it is not even possible to guarantee basic properties such as the eventual termination of cascaded rule firings. Complex rule programming is still considered an art, not science.

SQL is already a widely promulgated standard data access and manipulation language that is understood by hundreds of thousands of database programmers and is implemented by every serious DBMS in commercial use today, due to its combination of expressiveness and relative ease-of-use. Since StreamSQL inherits the basic syntax and semantics of SQL, it should be very familiar and attractive for people with SQL experience. On the other hand, rule languages have not yet been standardized with many alternative dialects offering (sometimes conflicting) syntax and semantics. Even though rule languages have gained good traction in a number of verticals, their adoption is still very limited compared to SQL-based systems.

Enabling High-Performance Stream Processing. SQL and StreamSQL are declarative languages that ask the programmer to answer only “what”, not “how”. This allows the underlying engine to use sophisticated optimization techniques (e.g., exploiting indexes, semantic rewrite rules, alternative operator implementations) to generate a highly efficient execution plan for a given user query. The major performance disadvantage of SQL relative to StreamSQL is that SQL must store streaming data before it can be processed, suffering a major overhead. On the other hand, StreamSQL implementations employ a “straight-through” processing model that allows it to avoid the storage overhead whenever possible.

Rule languages are also declarative in that they ask the programmer to specify only the condition-actions pairs and not how the conditions should be evaluated. In practice, sophisticated discrimination networks (e.g., Rete, TREAT) are used to significantly speed up the data-condition matching process at run time. It is important to note that the SQL-based execution plans can be linearized and thus run very efficiently with minimal overhead. Rule-based plans incur relatively higher run-time overhead; they cannot be linearized statically since matching is data-driven and needs to be performed at run-time. All approaches can, in principle, support parallel, multiprocessor operation, although this is more difficult to achieve for rule language due to the need to emulate globally shared memory.

	Rule Language	SQL	StreamSQL
Stream-oriented processing	Possible	Difficult	Yes
Streaming/stored data	No	Difficult	Yes
Data independence	No	Yes	Yes
Complex application support	Difficult	Possible	Possible
High performance	Difficult	Difficult	Possible

2.2 Tabular summary

We summarize the results of our discussion in the table above. Each entry in the table contains one of four values:

- **Yes:** The language naturally supports the feature.
- **No:** The language does not support the feature.
- **Possible:** The language *can* support the feature with trivial extensions. Check with a vendor for compliance.
- **Difficult:** The language *can* support the feature, but it is *difficult* due to the non-trivial modifications needed. Check with a vendor for compliance.

Overall, StreamSQL offers the best feature set for developing complex stream processing applications. This is not surprising, because StreamSQL has been designed and optimized from scratch to address the unique requirements and constraints of stream processing. Both SQL and rule languages were originally architected for a different class of applications with different underlying assumptions and requirements and, therefore, have fundamental constraints for this application domain.