Whitepaper

ARCADIA DATA

The State of Streaming Analytics

Enterprises are exploring a variety of architectures and technologies to incorporate real-time analytics on streaming data into their ecosystems. Such capabilities enable many valuable use cases: predictive maintenance, operations optimization, financial services risk reporting, cybersecurity, etc.

In this white paper, we explore three architectures that promote real-time analytics:

- Lambda Architecture
- Kappa Architecture
- Native Streaming

We will then review three popular open source technologies for SQL-on-streaming data:

- Apache Flink
- Apache Spark
- Apache Kafka

We will discuss programming challenges that arise and considerations that must be made in this new paradigm of streaming data analytics. Finally, we will cover several examples of streaming capabilities and use cases that are becoming mission-critical for modern enterprises.

Three Architectural Approaches to Real-Time

There are three primary architectures in use today to handle real-time streaming data. Each one has its advantages and disadvantages, so the right model will depend on your specific needs.

The Lambda Architecture

The Lambda Architecture was named and popularized by Nathan Marz in his seminal blog post, How to beat the CAP theorem.[1] In the Lambda Architecture, a real-time store and a batch processing store sit alongside each other. This lets you combine recent data with large-scale historical data in a unified interface. Lambda gives you access to all of your data, including your most recent data, without suffering from the data inconsistency that often plagues distributed systems.

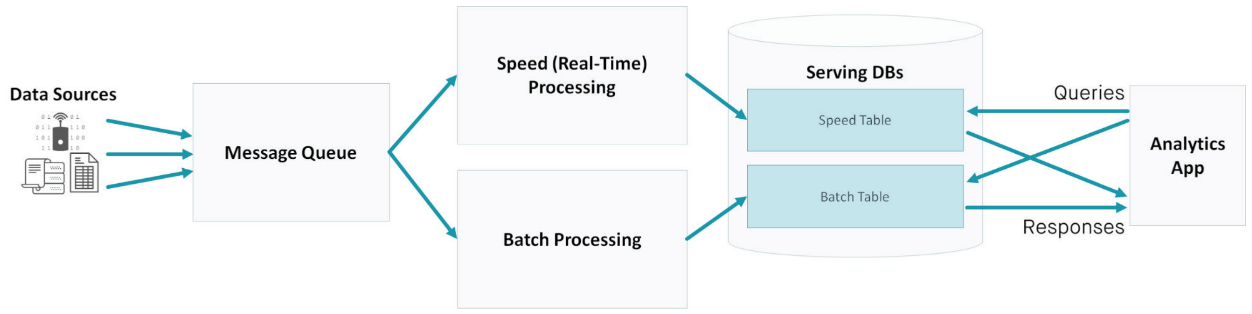
The CAP Theorem states that there is a trade-off between consistency and availability in distributed systems (assuming partition tolerance is a requirement), and Lambda attempts to overcome the CAP Theorem by reducing the challenge of maintaining consistency. One way to do that is to model data not as updatable facts, but rather as immutable, time-based facts. This means that a data point never changes; only new data points are created. For example, instead of saying that you moved from location A to location B, and thus your location "updated" from A to B, you can instead say at time T0 you were in location A, and at time T1 you are in location B. The new data point is "location B at time T1," while the old data point of "location A at time T0" remains valid

1 http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html

and immutable. In this paradigm, data is never changed and only created, so there is no opportunity for making inconsistent updates across replicas in a distributed system. Treating data as immutable facts is a good basis for streaming architectures.

This is implemented in Lambda by using separate stores for handling recent data and historical data. All data is treated as time-based, and is captured in full. Pre-computed tables and/or indexes can be created from the raw data in a batch process. This ensures order and avoids inconsistent updates that might result from having a distributed, updatable system.

Incoming data is sent to two separate processing engines that handle different time windows of data. The speed layer handles the most recent window of data, and processes the incoming data and stores it to a fast database (typically a NoSQL database). The batch layer handles all data that is not handled by the speed layer, and uses a batch engine like MapReduce or Spark to create pre-computed tables/indexes. The serving layer combines the batch layer and speed layer into a single interface.

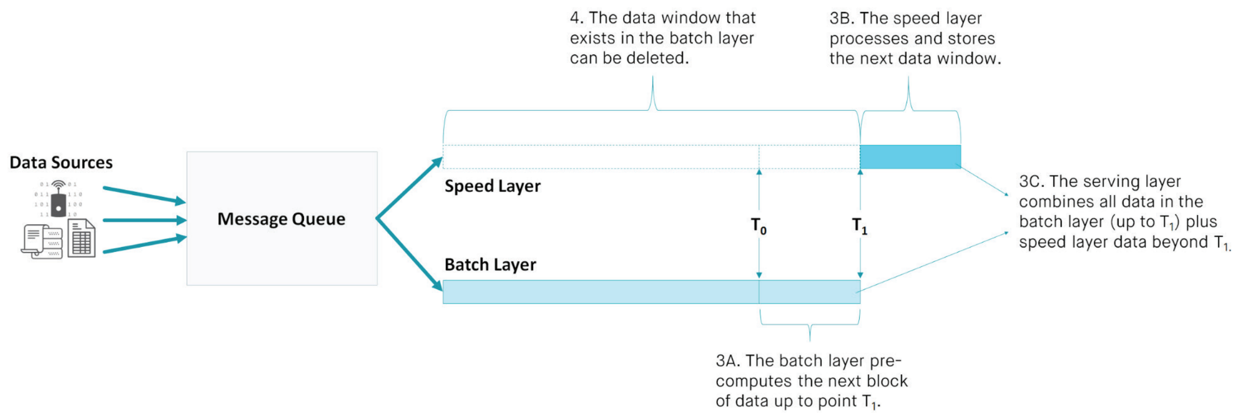


The Lambda Architecture combines real-time data with historical data.

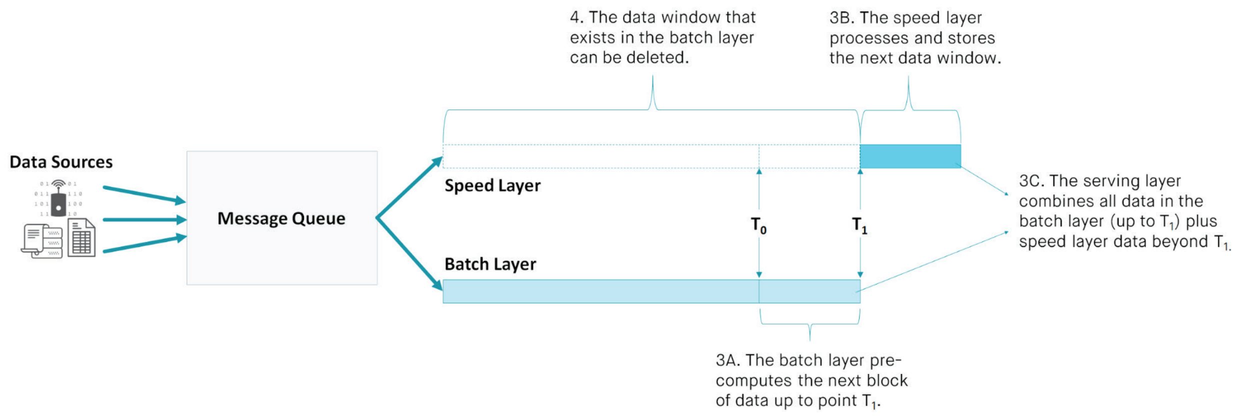
The process works as follows:

1. Initial index begins by sending the data stream to both the speed layer and the batch layer.
   A. The batch layer pre-computes all data. This means that all data up to the timestamp of the last data point, call it time T0, will be available for querying when the batch job is done.
   B. While the batch layer is running a pre-compute job, the speed layer is processing and storing the incoming stream. The speed layer makes all recent data not currently in the batch layer immediately available for querying.
   C. The serving layer will return query results from the batch layer combined with query results from the speed layer that do not include data that exists in the batch layer. For example, once the initial batch job is done, all data up to T0 will be served by the batch layer, and all data after T0 will be served by the speed layer.

2. The speed layer will eventually delete its set of data that is already pre-computed in the batch layer. In this example, all data prior to T0 will be deleted since it is available in the batch layer and thus not needed in the speed layer.


4. The data window that exists in the batch layer can be deleted.
3B. The speed layer processes and stores the next data window.
Data Sources
Message Queue
Speed Layer
Batch Layer
T_0
T_1
3C. The serving layer combines all data in the batch layer (up to T_1) plus speed layer data beyond T_1.
3A. The batch layer pre-computes the next block of data up to point T_1.

3. After the batch layer job completes, the indexing cycle repeats.
   A. The batch layer pre-computes the next window of data, say up to T1.
   B. The speed layer processes and stores the next window of data beyond T1.
   C. The serving layer returns query results for all batch layer data up to T1, and all speed layer data beyond T1.
4. Once that job is completed, the serving layer will serve data up to T1 from the batch layer and all data after T1 from the speed layer.


4. The data window that exists in the batch layer can be deleted.
3B. The speed layer processes and stores the next data window.
Data Sources
Message Queue
Speed Layer
Batch Layer
T_0
T_1
3C. The serving layer combines all data in the batch layer (up to T_1) plus speed layer data beyond T_1.
3A. The batch layer pre-computes the next block of data up to point T_1.

This window movement continues on, in which the batch layer indexing job lags behind the data stream, but the speed layer picks up the latest data not currently being indexed by the batch layer. Lambda does not specify the size of the real-time data window, so implementers are free to configure it as they see fit.

A key advantage of Lambda is human fault tolerance; in other words, its ability to overcome developer error. If any coding or modeling mistakes are made in either of the layers, the user-facing data can be re-computed from the raw data. This is also a key principle of managing big data, where you store all raw data so you can create new use cases from your valid, original data sets. All streaming architectures can benefit from this paradigm.

The main criticism of this approach is that the two stores require two distinct sets of programming logic. These code bases are difficult to keep in sync considering they are written on two different frameworks—Apache Storm and Apache Hadoop, for example. Lambda also does not define how the serving layer is implemented, which may lead to apprehension for anyone looking to implement Lambda since there is no clear guidance there. A limitation with Lambda is that the data in the speed layer is not truly real-time, as there is some latency associated with ingesting the data into the speed layer's store. Also, Lambda requires some overhead when querying the most recent time window of data, since all recent data is stored together, and queries must therefore include filtering to specify the data window.

Overall, the learning curve for getting Lambda up and running can be insignificant compared to the benefits, at least in the long term. Human fault tolerance as well as minimizing the complexity of distributed systems, while handling real-time and historical data simultaneously, are valuable advantages in a streaming and big data architecture.

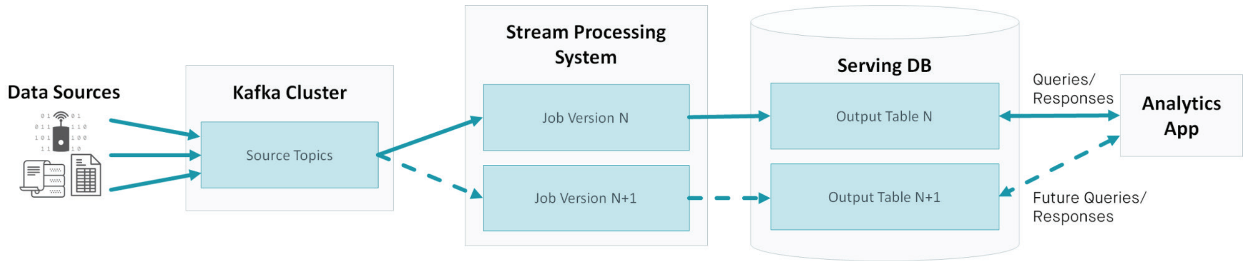
The Kappa Architecture

The Kappa Architecture was popularized by Jay Kreps as a response to the Lambda Architecture in his article, Questioning the Lambda Architecture.[2] With a Kappa Architecture, the principle of storing raw data as we saw in the Lambda Architecture is intact. The difference is the way in which the raw data is retained. In Kappa, raw data is first delivered to a publish-subscribe engine, namely Apache Kafka. Data sources known as "producers" will publish data to Kafka in an ordered manner. Since Kafka is designed for high-speed data ingestion, it is ideal for high volume and high velocity data environments like Internet of Things (IoT) analytics. Many producers can publish to Kafka simultaneously without the risk of overwriting data, which thus preserves data consistency. "Consumers" act as subscribers and read the data from Kafka, again in order, to process the data for other downstream usage.

In Kappa, the primary consumer is a stream processing engine like Apache Spark Streaming or Apache Flink, which uses low-level code to take some actions on the streaming data. Those actions involve data modeling so it can be stored in a high-speed serving database, typically a NoSQL database. From there, analytics can be run on the NoSQL database, which will have the most up-to-date view of the data. In a sense, Kappa simplifies Lambda by removing complexity of the distinct stores while also avoiding the consistency challenges of distributed systems. In the diagram below, the "Kafka cluster" is equivalent to the messaging queue in the Lambda Architecture.

2 https://www.oreilly.com/ideas/questioning-the-lambda-architecture

To handle human fault tolerance, historical data can be saved in Kafka and used as the source data for new processing code. Multiple jobs can be run on the source data, leading to multiple outputs. This is useful for resolving any problems with prior versions of code, but it can also be used for A-B testing of prior jobs with updated jobs. This human fault tolerance is accomplished with only one programming framework versus the two required in Lambda.



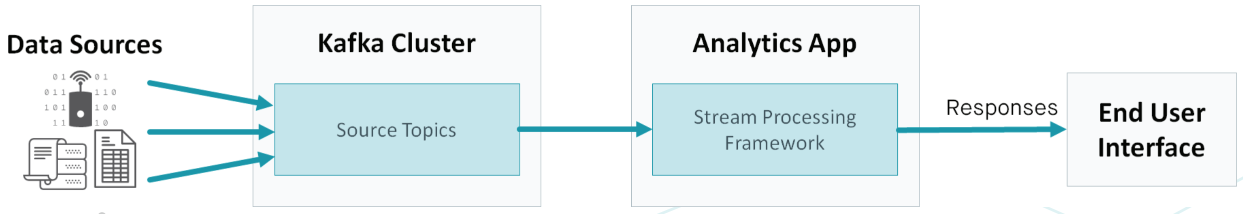
The Kappa Architecture uses the stream as the system of record.

The limitation of Kappa is that you do not get the complete history of raw data that Lambda provides for human fault tolerance. While some historical data can be stored in Kafka, there are limitations. In Kreps' article, he mentions 30 days as a historical time window, which is not nearly as long as the multi-year windows used by Lambda. This means that some combination of Lambda and Kappa is appropriate for streaming architectures that also integrate with historical data.

Native Streaming

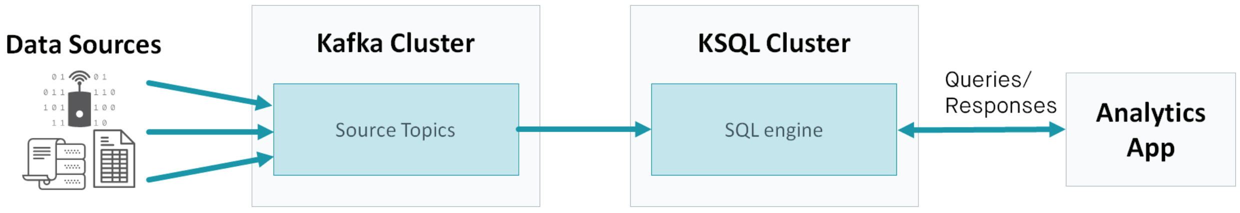
In a native streaming architecture, queries execute directly against the data in the streaming engine. So instead of creating an analytic data store on which queries are run, as in the architectures mentioned above, applications can act on each data point as they enter the system. This allows faster reaction times for real-time applications, especially those that are alert-based, which is especially valuable for use cases such as predictive maintenance, assembly line quality control, and operations optimization.

This architecture is typically handled by a stream processing framework like Apache Spark Streaming, Apache Flink, or Apache Storm running on data in Apache Kafka. Similar to the architectures above, the queries are run from custom applications that are built by developers. Since the queries are built into the analytics application, there often is no end-user interaction. Rather, the applications take some action based on trends, patterns, and/or specific events in the data.



Native streaming allows the lowest latency in real-time analytics.

An emerging enhancement to this architecture entails the use of SQL as the query mechanism for data in Apache Kafka. For example, Confluent's KSQL provides a SQL engine that directly queries the data stream, allowing developers to use the popular SQL language in their streaming analytics applications. Since the ubiquitous SQL language is involved, this architecture has the potential to open up real-time streaming data to a wide variety of existing SQL-based tools, and subsequently a wider variety of end users.



SQL-on-streaming data can make streaming architectures more mainstream.

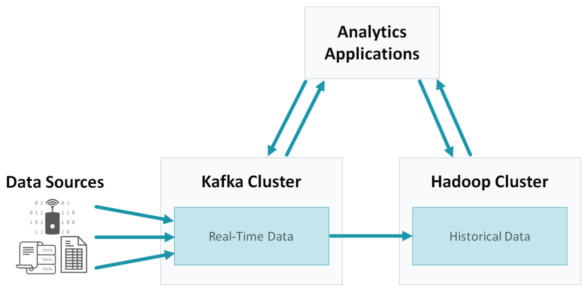
Benefits and Challenges of the Architectures

The table below summarizes the key strengths and weaknesses of these three architectures.

	Strengths	**Weaknesses**
Lambda	• Analysis on large historical data sets • Human fault tolerance for pre-computed results.	• Lack of ad-hoc queries. Data models are pre-built, so the queries you can run are limited. • Duplicated logic creates a challenge, as applications are built on top of the two stores. • No implementation details on how two stores are synchronized. • Increased administration.
Kappa	• Applications only need one set of logic. • Human fault tolerance for pre-computed results on smaller windows of data. • Simplified stack.	• Some limits to ad-hoc queries. • May still need to maintain two systems with separate security models.
Native Streaming	• Very low latency analytics ("true" real-time). • Ad-hoc real-time queries. • Simplified stack. • Support for SQL, allowing more tools and users. • Designed for a push-based model rather than a polling model, so you can get notifications on continuous event streams without re-running queries.	• New technology (challenges are discussed in more detail later in the section Using SQL on Streams).

Note that some of the weaknesses of the Lambda Architecture are due to the fact that it covers both real-time and historical data fairly well. This dual focus naturally will add complexity and challenges. If real-time data is your only concern, then the Kappa Architecture or the native streaming architecture might be ideal for you.

However, most streaming environments (and big data environments) should retain longer periods of data than Kafka can ideally store. This means a combination of the architectures above using Hadoop and Kafka might be a good approach, as shown in the diagram below. This approach does not solve the challenge of querying all data in real time, so that remains the trade-off here. In situations where real-time data is always queried separately from historical data, this approach makes sense.



Store streaming data first in Kafka, then save it as historical data in Hadoop.

Technologies for SQL-on-Streams

Now that we have described the basic architectures for capturing and querying data streams, let's focus on the third architecture. We'll look at the popular technologies for using SQL or some "SQL-like" query language on streams. For this white paper, we describe Apache Flink, Apache Spark, and Confluent's KSQL, three open source stream processing frameworks. While this certainly is not an exhaustive list, it does allow us to compare the SQL-readiness of three frameworks that are in various stages of maturity and have active development communities.

Apache Flink

Apache Flink is the oldest of the three SQL-on-streams technologies presented here, with its first pre-Apache release dating back to May 2011.[3] Apache Flink features the Table API for stream processing and SQL batch processing. The Table API lets you use Scala and Java to write select-, filter-, and join-style queries. SQL support is based on the Apache Calcite framework for databases and data management.

Queries have the same semantics and return the same results whether they are written with the Table API or SQL, and whether the inputs are batch or stream. Despite, or because of, this tight integration, neither the Table API nor SQL are complete. Not all operations allow every combination of query type and input.

Many companies, software projects, and research institutes use Apache Flink.[4]

3 https://en.wikipedia.org/wiki/Apache_Flink#History

4 https://cwiki.apache.org/confluence/display/FLINK/Powered+by+Flink

Apache Spark

Apache Spark is an open source distributed computing framework that has rapidly grown in popularity since it was first released in 2012.[5] Spark Streaming is the module of Spark that ingests data in micro-batches, and the same application code can be used for both streaming and batch analytics. This makes it easy to implement the Lambda Architecture covered earlier. However, because of the use of micro-batches, "real-time" is actually determined by the latency of the micro-batch.

Spark has always been considered easy to use, and ongoing work is making Spark even easier. For example, in Spark 2.0, developers added Structured Streaming, a higher-level API for building consistent, fault-tolerant streaming applications that integrate with storage, serving systems, and batch jobs.[6]

The Spark SQL engine is based on the Catalyst optimizer,[7] which is designed to make it easy to add new optimizations and for developers to extend the optimizer.[8] In fact, this development community has proved a potent ingredient for the popularity of Spark. A market research survey sponsored by big data vendor Cloudera in late 2016 showed that real-time stream processing was the second-most reported use case for Spark.[9]

KSQL

KSQL is an open source stream processing platform created by Confluent, the software company founded by the creators of Apache Kafka. KSQL is one of the newest SQL-on-streams technologies.[10]

KSQL uses SQL-like semantics to read, write, and process streaming data in the form of Kafka topics as an alternative to writing in programming languages such as Java or Python. KSQL supports joins, aggregations, event-time windowing, sessionization, and more. KSQL can "listen" to a Kafka topic to receive new data and respond to that data. It includes a familiar command-line interface that acts as a client to the KSQL server, which runs the engine that executes KSQL queries.[11] KSQL uses the Kafka Streams API to issue queries and consume results, which provides multiple benefits including simplified integrations.[12]

5 http://spark.apache.org/powered-by.html
6 https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html
7 https://www.slideshare.net/databricks/a-deep-dive-into-spark-sqls-catalyst-optimizer-with-yin-huai
8 https://www.packtpub.com/mapt/book/big_data_and_business_intelligence/9781783987061/4/ch04lvl1sec31/understanding-the-catalyst-optimizer
9 http://tanejagroup.com/profiles-reports/request/apache-spark-market-survey-cloudera-sponsored-research#.WhUC7RNSzBJ
10 https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines
11 https://github.com/confluentinc/ksql/blob/0.1.x/docs/concepts.md#concepts
12 https://kafka.apache.org/documentation/streams/

Using SQL-on-Streams

In an ideal world, SQL is the same no matter where you run it. In practice, however, this is not true. There are many variants of SQL and SQL-like languages used in modern enterprise architectures. The potential for variation—and new concepts that must be addressed—further expands when you use continuous queries to capture, interact with, and respond to real-time streaming data. So while the use of SQL on streaming data has a relatively low learning curve due to the familiarity of core SQL, some extensions need to be explored first.

Windows and Aggregates

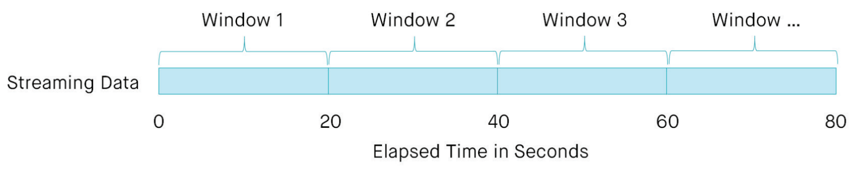
Let's consider windows and aggregates. In a traditional relational database, you typically perform an aggregate query such as a "group by" statement over a finite number of rows. With streaming data, however, the data available to query frequently changes. Data is regularly being ingested into the system, and you may be off-loading or dumping some historical or dated data.

Therefore, when building your application, you must consider how you want aggregations to operate. When new data arrives, does it invalidate your previous result? How does the user know the result is correct?

To account for this in streaming analytics, queries are often run over a window, a sliding time-based subset of your data. A window on a stream is different from a window function in the traditional SQL sense. The former is a specified time period, and the latter is a specified subset of rows. When you specify a window on a stream, you are requesting a result from the specified time period. Once that time period passes, the query proceeds to the next set of results.

The SQL snippets below show three examples of how windows are defined in SQL-on-streams. Each of these sample SQL-on-streams queries aggregates the quantity of an item within 20- second periods.

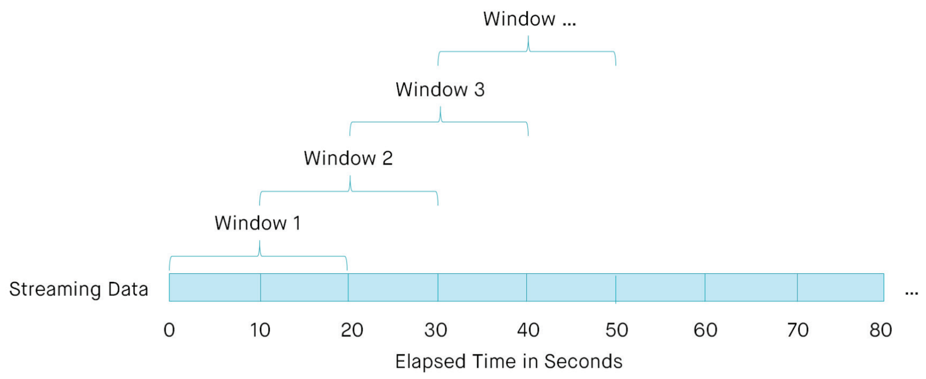
Window tumbling segments streams into distinct segments that do not overlap.



```
SELECT item_id, SUM(quantity)
FROM orders
WINDOW TUMBLING (SIZE 20 SECONDS)
GROUP BY item_id;
```

Tumbling windows are contiguous and non-overlapping.

Hopping windows are similar to tumbling windows, but they advance by a specified period so they can allow some overlapping.

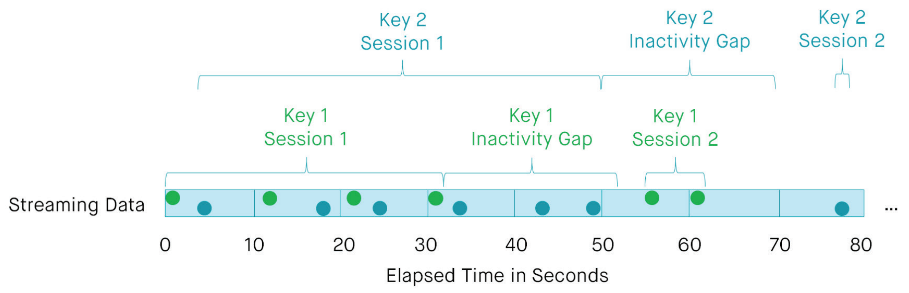


```
SELECT item_id, SUM(quantity)
FROM orders
WINDOW HOPPING (SIZE 20 SECONDS, ADVANCE BY 10 SECONDS)
GROUP BY item_id;
```

Hopping windows advance by a specified time and can overlap.

Session windows aggregate data based on key-based events rather than fixed time periods. Session windows are dynamic in size and start time, and are correlated with row keys. A session is defined by a grouping of events with the same key. A session "ends" when there is a period of inactivity (i.e., the "inactivity gap" specified in the query) in which no events with that same key appears. This type of windowing is useful for cases like user behavior analysis on web pages, in which the inactivity gap suggests a given user has left the site.

In the diagram below, note that sessions for each key, with events depicted by blue and green circles, continue to grow until an inactivity gap appears. The next event with that same key starts a new session for that key.



```
SELECT item_id, SUM(quantity)
FROM orders
WINDOW SESSION (20 SECONDS)
GROUP BY item_id;
```

Session windows are defined by groupings of key-based events.

Continuous Queries

In another radical departure from traditional SQL, when you issue a SQL-on-streams query, the query never completes and returns a final result. You are continuously receiving results. This creates a new paradigm for application development and lifecycle management. With a single query, you will continue to get new results as you stream new data (or cease to stream data).

Joins

With streams, the possibilities for joins quickly multiply beyond traditional relational database joins. For example, you can join two streams, you can join a stream to an arbitrary data source, and you can join a stream with a traditional table to augment the table or the stream. Traditional database concepts like starflake schema, dimensional queries, and fact tables are all technically feasible with streams, though the overhead of those concepts becomes more complex.

Currently, join support for streams varies quite a bit among SQL-on-streams technologies. Depending on the framework, the capability may be non-existent, or you may be able to create joins between two streams, or create joins between a stream and a static table. There is significant ongoing development in this area for many of the SQL-on-streams technologies.

Streaming Capabilities and Use Cases

Ultimately, the usefulness of streaming technologies will be measured by the businesses who depend on them for critical capabilities and use cases. Here are a few examples of real-time capabilities and the use cases they enable that are critical for modern enterprises:

Alert Responses

- A real-time machine learning or alerting system notices a situation and issues an alert or incident for your subject matter expert to investigate.
- The user may want a real-time dashboard that continuously displays what is happening for use cases such as cybersecurity and healthcare monitoring.

Side-by-Side Historical and Real-Time Analysis

- Users are looking through deep historical information with traditional OLAP techniques and find something interesting. They then want to pivot into a real-time view of the data to test their theory. For example, they may be examining a malfunctioning device, an underperforming marketing campaign, or fraud at an ATM.

Stream Data Enrichment

- You may want to join a data stream to an existing table to get complete information about the data in the stream. In a predictive maintenance use case, for example, you may want to join "machine_id" in the stream to a lookup table to be able to view all available data about your machines.

Streaming visualizations are especially becoming critical to a variety of use cases. In predictive maintenance, visualizations can help identify trends and patterns that reveal impending failures in equipment. In financial services risk reporting, streaming visualizations can provide an immediate understanding of capital market risk so you can make quick decisions about asset allocations and continuously ensure compliance. For cybersecurity, streaming visualizations help detect and provide alerts on anomalous behavior in network log data in real time for a wider variety of end users, not just data scientists.

Conclusion

We are in a time of unprecedented interest and development around streaming technologies and analytics. This is driven both by technological advancements and the increased realization of business value from streaming analytics and real-time visual analysis. Expect this trend to continue at an even faster pace as we adopt more connected devices, and the connected world built on the Internet of Things matures. Businesses looking for the next source of competitive advantage will turn to streaming data to gain insights that they cannot get from their existing approaches to analytics.

In addition to the continued maturation of SQL-on-streams technologies, visualization tools for non-technical users will add new capabilities to take advantage of real-time streams. In most cases, existing tools in the market will need a significant architectural overhaul to properly handle streams. On the other hand, visualization technologies designed for big data are prepared for this new data source. One example is Arcadia Data. By providing native connectivity to engines like KSQL and also providing real-time screen refreshes and the ability to pause/play visualizations on the streaming data, Arcadia Data enables business analysts and business users to take full advantage of streaming data.

Contact Us
Arcadia Data
999 Baker Way, Suite 120
San Mateo, CA 94404

Phone: (415) 680-3535
Email: info@arcadiadata.com