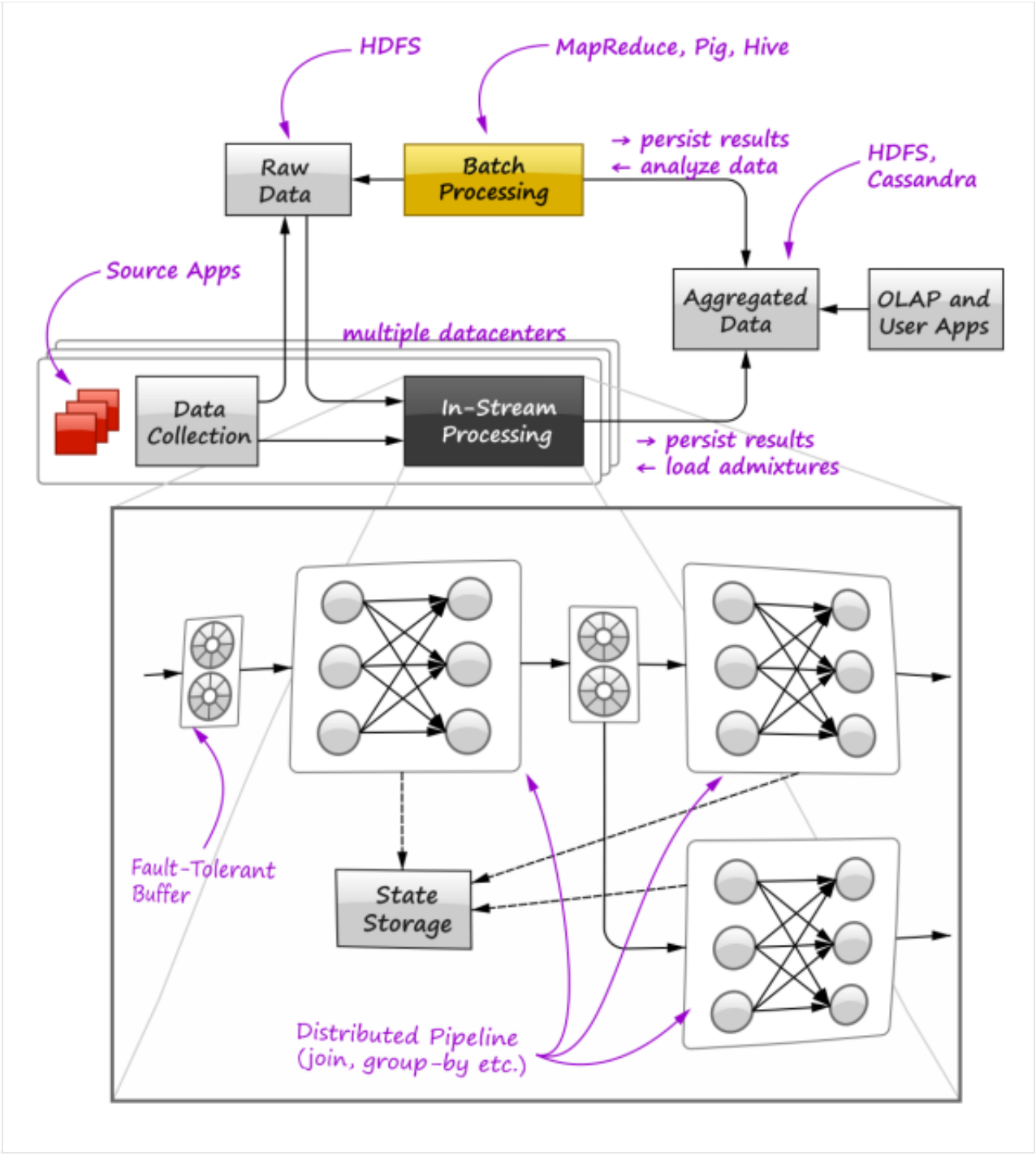
In-Stream Big Data Processing

Articles on Big Data, NoSQL, and Highly Scalable Software Engineering

The shortcomings and drawbacks of batch-oriented data processing were widely recognized by the Big Data community quite a long time ago. It became clear that real-time query processing and in-stream processing is the immediate need in many practical applications. In recent years, this idea got a lot of traction and a whole bunch of solutions like Twitter's Storm, Yahoo's S4, Cloudera's Impala, Apache Spark, and Apache Tez appeared and joined the army of Big Data and NoSQL systems. This article is an effort to explore techniques used by developers of in-stream data processing systems, trace the connections of these techniques to massive batch processing and OLTP/OLAP databases, and discuss how one unified query engine can support in-stream, batch, and OLAP processing at the same time.

At Grid Dynamics, we recently faced a necessity to build an in-stream data processing system that aimed to crunch about 8 billion events daily providing fault-tolerance and strict transactioanlity i.e. none of these events can be lost or duplicated. This system has been designed to supplement and succeed the existing Hadoop-based system that had too high latency of data processing and too high maintenance costs. The requirements and the system itself were so generic and typical that we describe it below as a canonical model, just like an abstract problem statement.

A high-level overview of the environment we worked with is shown in the figure below:



One can see that this environment is a typical Big Data installation: there is a set of applications that produce the raw data in multiple datacenters, the data is shipped by means of Data Collection subsystem to HDFS located in the central facility, then the raw data is aggregated and analyzed using the standard Hadoop stack (MapReduce, Pig, Hive) and the aggregated results are stored in HDFS and NoSQL, imported to the OLAP database and accessed by custom user applications. Our goal was to equip all facilities with a new in-stream engine (shown in the bottom of the figure) that processes most intensive data flows and ships the pre-aggregated data to the central facility, thus decreasing the amount of raw data and heavy batch jobs in Hadoop. The design of the in-stream processing engine itself

was driven by the following requirements:

- SQL-like functionality. The engine has to evaluate SQL-like queries continuously, including joins over time windows and different aggregation functions that implement quite complex custom business logic. The engine can also involve relatively static data (admixtures) loaded from the stores of Aggregated Data. Complex multi-pass data mining algorithms are beyond the immediate goals.
- Modularity and flexibility. It is not to say that one can simply issue a SQL-like query and the corresponding pipeline will be created and deployed automatically, but it should be relatively easy to assemble quite complex data processing chains by linking one block to another.
- Fault-tolerance. Strict fault-tolerance is a principal requirement for the engine. As it sketched in the bottom part of the figure, one possible design of the engine is to use distributed data processing pipelines that implement operations like joins and aggregations or chains of such operations, and connect these pipelines by means of fault-tolerant persistent buffers. These buffers also improve modularity of the system by enabling publish/subscribe communication style and easy addition/removal of the pipelines. The pipelines can be stateful and the engine's middleware should provide a persistent storage to enable state checkpointing. All these topics will be discussed in the later sections of the article.
- Interoperability with Hadoop. The engine should be able to ingest both streaming data and data from Hadoop i.e. serve as a custom query engine atop of HDFS.
- High performance and mobility. The system should deliver performance of tens of thousands messages per second even on clusters of minimal size. The engine should be compact and efficient, so one can deploy it in multiple datacenters on small clusters.

To find out how such a system can be implemented, we discuss the following topics in the rest of the article:

- First, we explore relations between in-stream data processing systems, massive batch processing systems, and relational query engines to understand how in-stream processing can leverage a huge number of techniques that were devised for other classes of systems.
- Second, we describe a number of patterns and techniques that are frequently used in building of in-stream processing frameworks and systems. In addition, we survey the current and emerging technologies and provide a few implementation tips.

The article is based on a research project developed at Grid Dynamics Labs. Much of the credit goes to Alexey Kharlamov and Rafael Bagmanov who led the project and other contributors: Dmitry Suslov, Konstantine Golikov, Evelina Stepanova, Anatoly Vinogradov, Roman Belous, and Varvara Strizhkova.

Basics of Distributed Query Processing

It is clear that distributed in-stream data processing has something to do with query processing in distributed relational databases. Many standard query processing techniques can be employed by in-stream processing engine, so it is extremely useful to understand classical algorithms of distributed query processing and see how it all relates to in-stream processing and other popular paradigms like MapReduce.

Distributed query processing is a very large area of knowledge that was under development for decades, so we start with a brief overview of the main techniques just to provide a context for further discussion.

Partitioning and Shuffling

Distributed and parallel query processing heavily relies on data **partitioning** to break down a large data set into multiple pieces that can be processed by independent processors. Query processing could consist of multiple steps and each step could require its own partitioning strategy, so data **shuffling** is an operation frequently performed by distributed databases.

Although optimal partitioning for selection and projection operations can be tricky (e.g. for range queries), we can assume that for in-stream data filtering it is practically enough to distribute data among the processors using a hash-based partitioning.

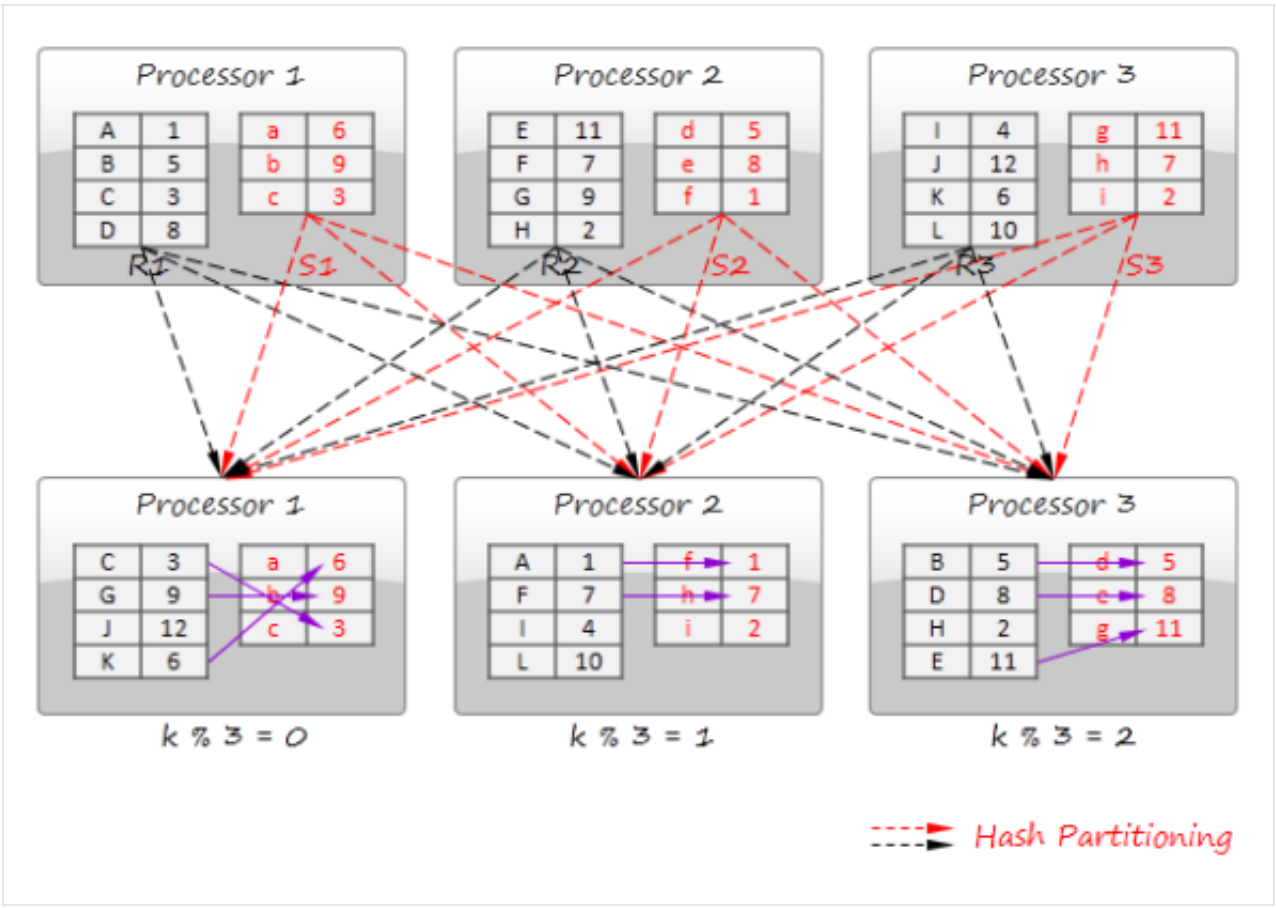
Processing of distributed joins is not so easy and requires a more thorough examination. In distributed environments, parallelism of join processing is achieved through data partitioning, i.e. the data is distributed among processors and each processor employs a serial join algorithm (e.g. nested-loop join or sort-merge join or hash-based join) to process its part of the data. The final results are consolidated from the results obtained from different processors.

There are two main data partitioning techniques that can be employed by distributed join processing:

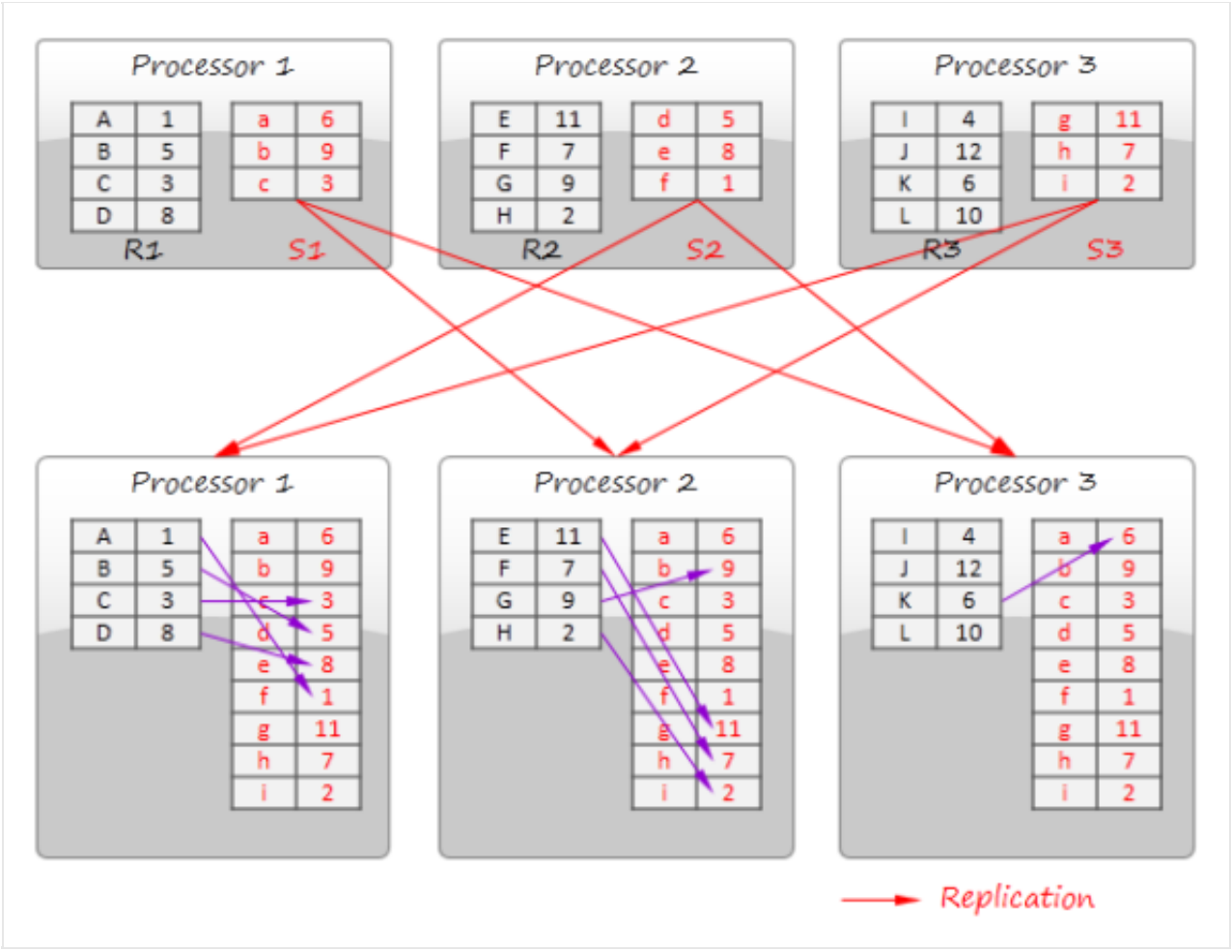
- Disjoint data partitioning

- Divide and broadcast join

Disjoint data partitioning technique shuffles the data into several partitions in such a way that join keys in different partitions do not overlap. Each processor performs the join operation on each of these partitions and the final result is obtained as a simple concatenation of the results obtained from different processors. Consider an example where relation R is joined with relation S on a numerical key k and a simple modulo-based hash function is used to produce the partitions (it is assumes that the data initially distributed among the processors based on some other policy):

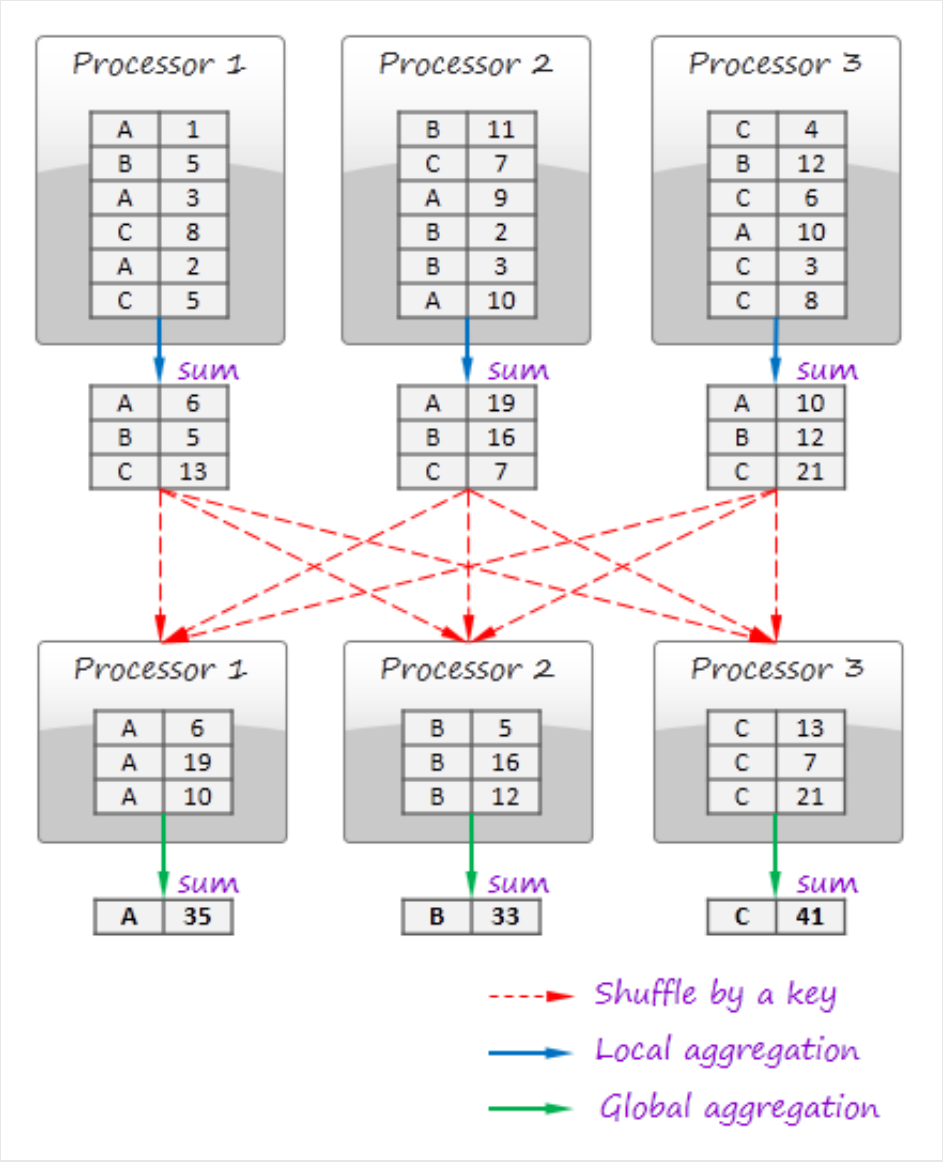


The divide and broadcast join algorithm is illustrated in the figure below. This method divides the first data set into multiple disjoint partitions (R1, R2, and R3 in the figure) and replicates the second data set to all processors. In a distributed database, division typically is not a part of the query processing itself because data sets are initially distributed among multiple nodes.



This strategy is applicable for joining of a large relation with a small relation or two small relations. In-stream data processing systems can employ this technique for stream enrichment i.e. joining a static data (admixture) to a data stream.

Processing of GroupBy queries also relies on shuffling and fundamentally similar to the MapReduce paradigm in its pure form. Consider an example where the data is grouped by a string key and sum of the numerical values is computed in each group:



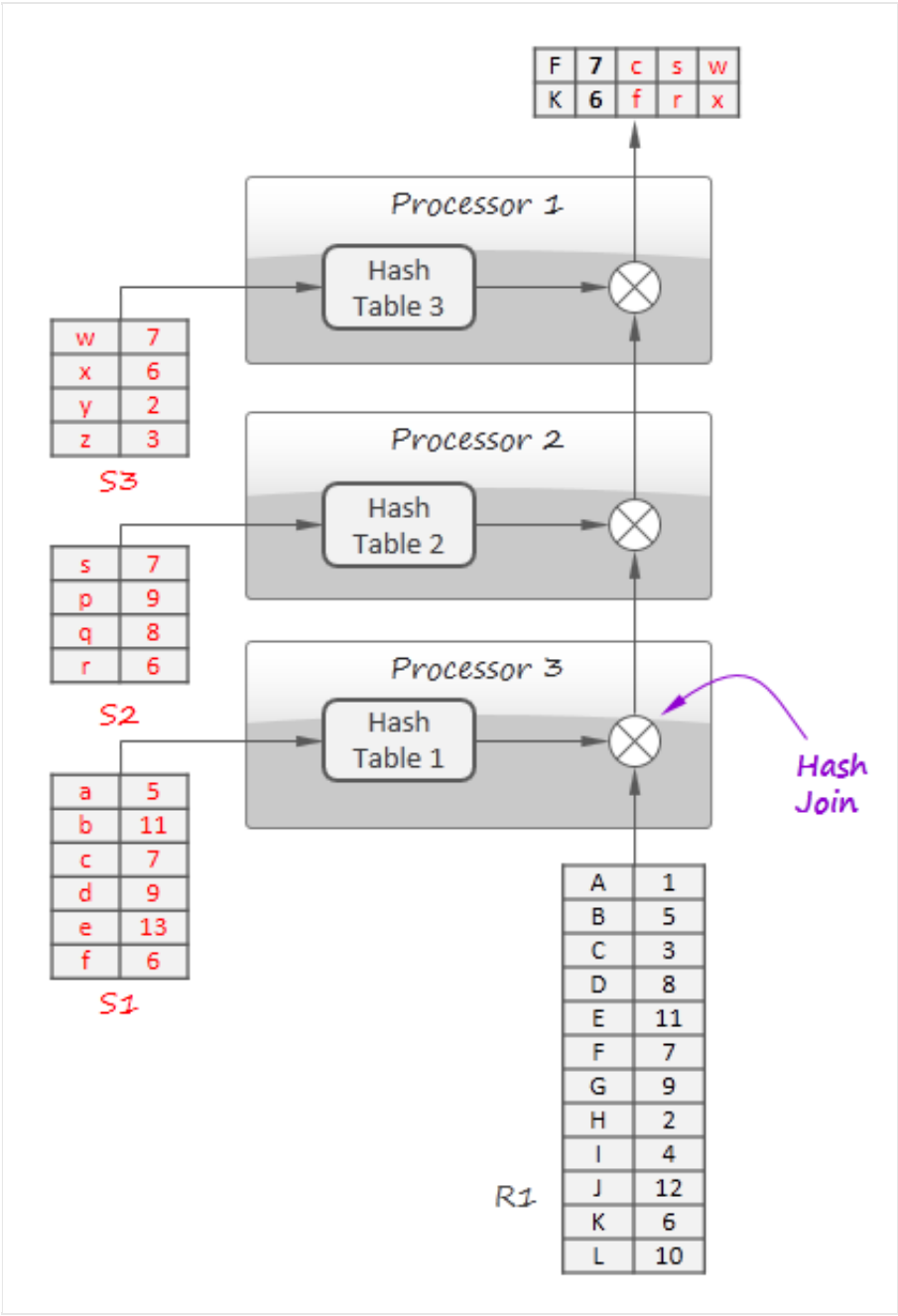
In this example, computation consists of two steps: local aggregation and global aggregation. These steps basically correspond to Map and Reduce operations. Local aggregation is optional and raw records can be emitted, shuffled, and aggregated on a global aggregation phase.

The whole point of this section is that all the algorithms above can be naturally implemented using a message passing architectural style i.e. the query execution engine can be considered as a distributed network of nodes connected by the messaging queues. It is conceptually similar to the in-stream processing pipelines.

Pipelining

In the previous section, we noted that many distributed query processing algorithms resemble message passing networks. However, it is not enough to organize efficient in-stream processing: all operators in a query should be chained in such a way that the data

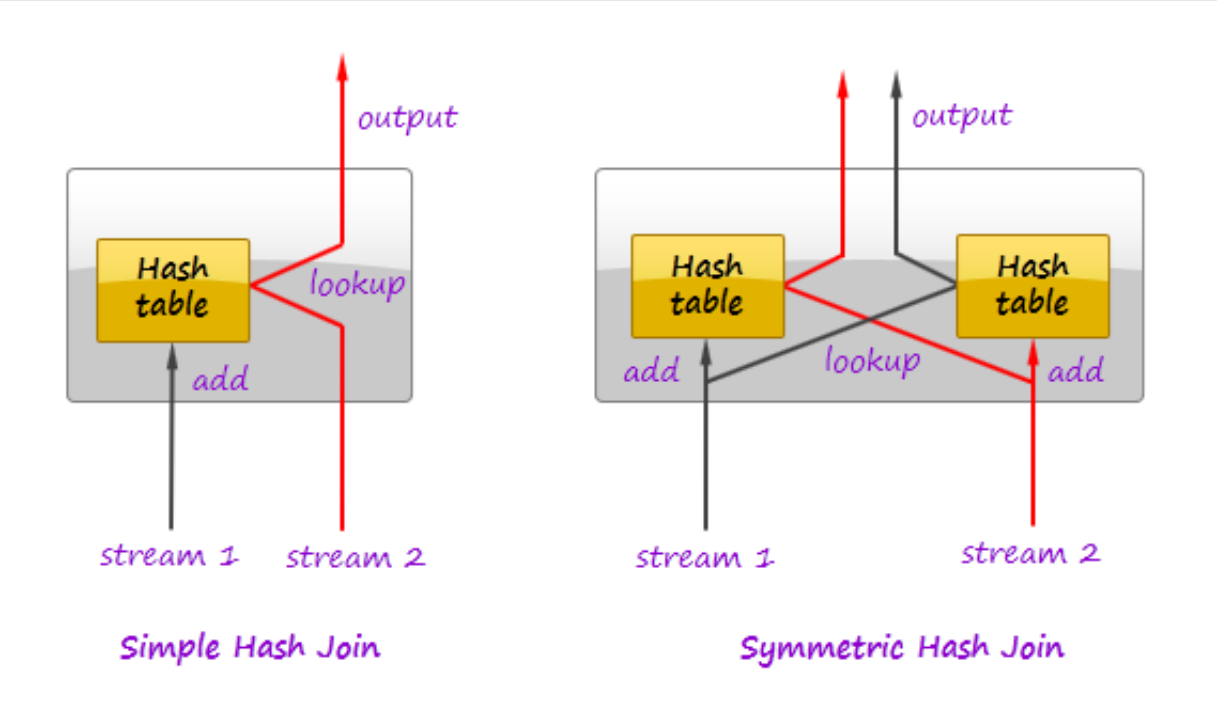
flows smoothly through the entire pipeline i.e. neither operation should block processing by waiting for a large piece of input data without producing any output or by writing intermediate results on disk. Some operations like sorting are inherently incompatible with this concept (obviously, a sorting block cannot produce any output until the entire input is ingested), but in many cases pipelining algorithms are applicable. A typical example of pipelining is shown below:



In this example, the hash join algorithm is employed to join four relations: R1, S1, S2, and S3 using 3 processors. The idea is to build hash tables for S1, S2 and S3 in parallel and then stream R1 tuples one by one though the pipeline that joins them with S1, S2 and S3 by looking up matches in the hash tables. In-stream processing naturally employs this

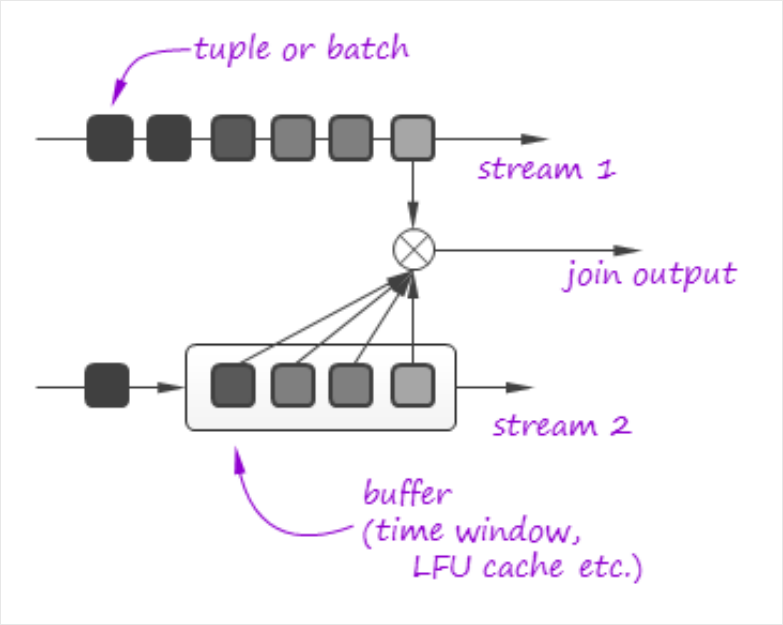
technique to join a data stream with the static data (*admixtures*).

In relational databases, join operation can take advantage of pipelining by using the symmetric hash join algorithm or some of its advanced variants [1,2]. Symmetric hash join is a generalization of hash join. Whereas a normal hash join requires at least one of its inputs to be completely available to produce first results (the input is needed to build a hash table), symmetric hash join is able to produce first results immediately. In contrast to the normal hash join, it maintains hash tables for both inputs and populates these tables as tuples arrive:



As a tuple comes in, the joiner first looks it up in the hash table of the other stream. If match is found, an output tuple is produced. Then the tuple is inserted in its own hash table.

However, it does not make a lot of sense to perform a complete join of infinite streams. In many cases join is performed on a finite time window or other type of buffer e.g. LFU cache that contains most frequent tuples in the stream. Symmetric hash join can be employed if the buffer is large comparing to the stream rate or buffer is flushed frequently according to some application logic or buffer eviction strategy is not predictable. In other cases, simple hash join is often sufficient since the buffer is constantly full and does not block the processing:



It is worth noting that in-stream processing often deals with sophisticated stream correlation algorithms where records are matched based on scoring metrics, not on field equality condition. A more complex system of buffers can be required for both streams in such cases.

In-Stream Processing Patterns

In the previous section, we discussed a number of standard query processing techniques that can be used in massively parallel stream processing. Thus, **on a conceptual level, an efficient query engine in a distributed database can act as a stream processing system and vice versa, a stream processing system can act as a distributed database query engine. Shuffling and pipelining are the key techniques of distributed query processing and message passing networks can naturally implement them.** However, things are not so simple. In a contrast to database query engines where reliability is not critical because a read-only query can always be restarted, streaming systems should pay a lot of attention to reliable events processing. In this section, we discuss a number of techniques that are used by streaming systems to provide message delivery guarantees and some other patterns that are not typical for standard query processing.

Stream Replay

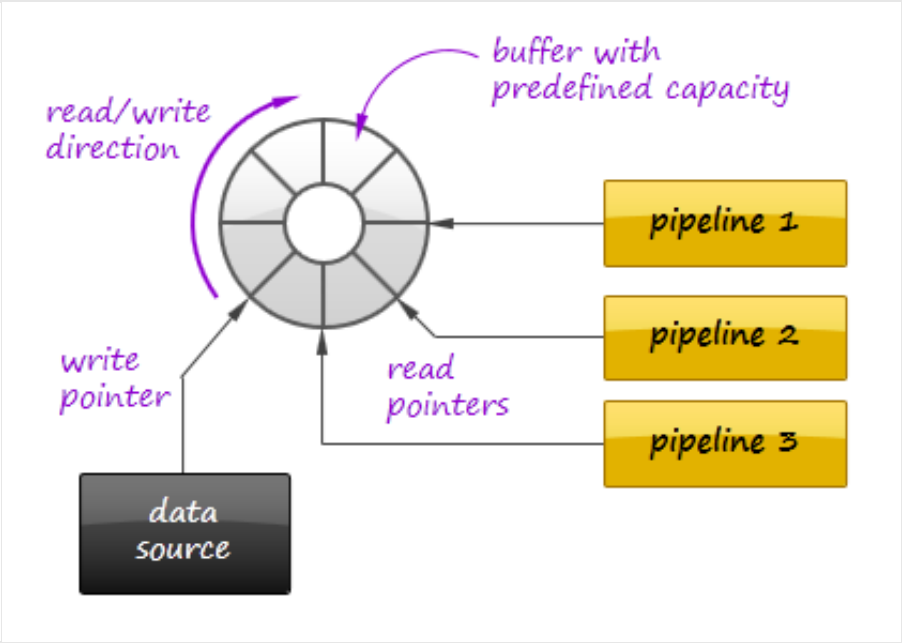
Ability to rewind data stream back in time and replay the data is very important for in-stream processing systems because of the following reasons:

- This is the only way to guarantee correct data processing. Even if data processing

pipeline is fault-tolerant, it is very problematic to guarantee that the deployed processing logic is defect-free. One can always face a necessity to fix and redeploy the system and replay the data on a new version of the pipeline.

- Issue investigation could require ad hoc queries. If something goes wrong, one could need to rerun the system on the problematic data with better logging or with code alternations.
- Although it is not always the case, the in-stream processing system can be designed in such a way that it re-reads individual messages from the source in case of processing errors and local failures, even if the system in general is fault-tolerant.

As a result, the input data typically goes from the data source to the in-stream pipeline via a persistent buffer that allows clients to move their reading pointers back and forth.



Kafka messaging queue is well known implementation of such a buffer that also supports scalable distributed deployments, fault-tolerance, and provides high performance.

As a bottom line, Stream Replay technique imposes the following requirements of the system design:

- The system is able to store the raw input data for a preconfigured period time.
- The system is able to revoke a part of the produced results, replay the corresponding input data and produce a new version of the results.
- The system should work fast enough to rewind the data back in time, replay them, and then catch up with the constantly arriving data stream.

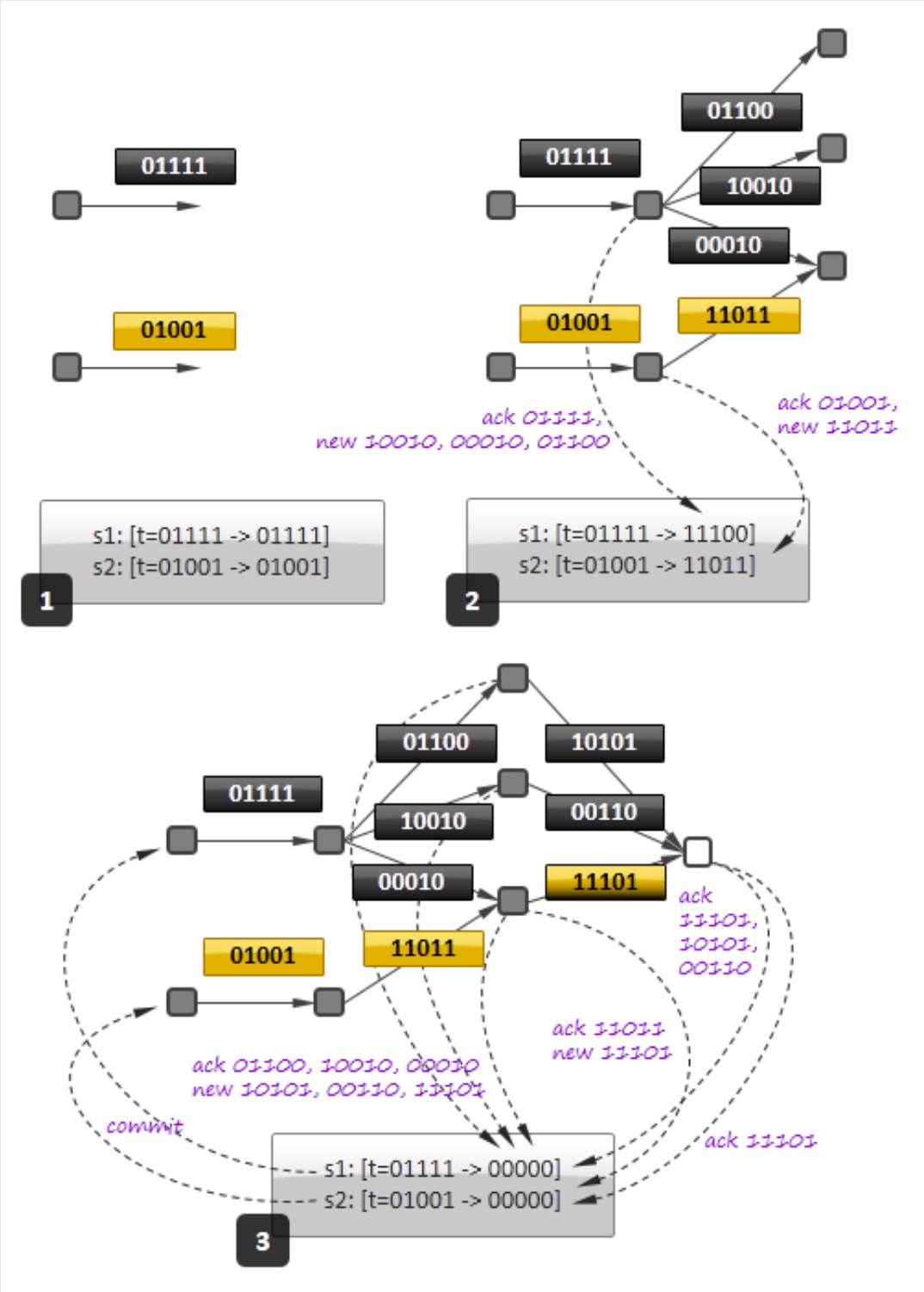
Lineage Tracking

In a streaming system, events flow through a chain of processors until the result reaches the final destination (like an external database). Each input event produces a directed graph of descendant events (lineage) that ends by the final results. To guarantee reliable data processing, it is necessary to ensure that the entire graph was processed successfully and to restart processing in case of failures.

Efficient lineage tracking is not a trivial problem. Let us first consider how Twitter's Storm tracks the messages to guarantee at-least-once delivery semantics (see the diagram below):

- All events that emitted by the sources (first nodes in the data processing graph) are marked by a random ID. For each source, the framework maintains a set of pairs [event ID -> signature] for each initial event. The signature is initially initialized by the event ID.
- Downstream nodes can generate zero or more events based on the received initial event. Each event carries its own random ID and the ID of the initial event.
- If the event is successfully received and processed by the next node in the graph, this node updates the signature of the corresponding initial event by XORing the signature with (a) ID of the incoming event and (b) IDs of all events produced based on the incoming event. In the part 2 of diagram below, event 01111 produces events 01100, 10010, and 00010, so the signature for event 01111 becomes 11100 (= 01111 (initial value) xor 01111 xor 01100 xor 10010 xor 00010).
- An event can be produced based on more than one incoming event. In this case, it is attached several initial event and carries more than one initial IDs downstream (yellow-black event in the part 3 of the figure below).
- The event considered to be successfully processed as soon as its signature turns into zero i.e. the final node acknowledged that the last event in the graph was processed successfully and no events were emitted downstream. The framework sends a commit message to the source node (see part 3 in the diagram below).
- The framework traverses a table of the initial events periodically looking for old uncommitted events (events with non-zero signature). Such events are considered as failed and the framework asks the source nodes to replay them.
- It is important to note that the order of signature updates is not important due to commutative nature of the XOR operation. In the figure below, acknowledgements depicted in the part 2 can arrive after acknowledgements depicted in the part 3. This enables fully asynchronous processing.
- One can note that the algorithm above is not strictly reliable – the signature could turn into zero accidentally due to unfortunate combination of IDs. However, 64-bit

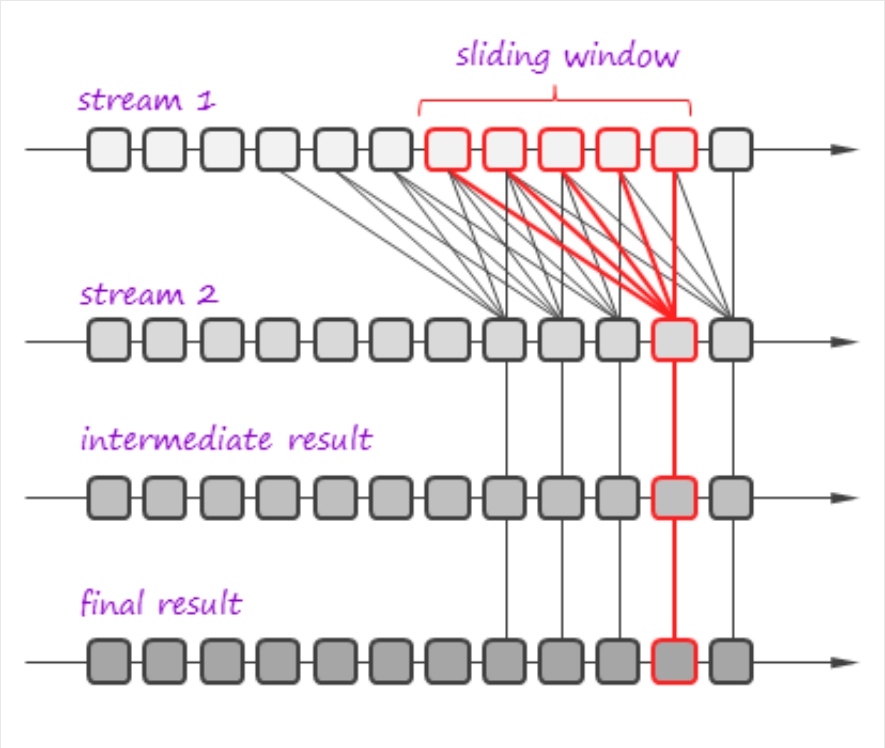
IDs are sufficient to guarantee a very low probability of error, about 2^(-64), that is acceptable in almost all practical applications. As result, the table of signatures could have a small memory footprint.



The described approach is elegant due to its decentrilized nature: nodes act independently sending acknowledgement messages, there is no cental entity that tracks all lineages explicitly. However, it could be difficult to manage transactional processing in this way for flows that maintain sliding windows or other buffers. For example, processing on a sliding

window can involve hundreds of thousands events at each moment of time, so it becomes difficult to manage acknowledgements because many events stay uncommitted or computational state should be persisted frequently.

An alternative approach is used in Apache Spark [3]. The idea is to consider the final result as a function of the incoming data. To simplify lineage tracking, the framework processes events in batches, so the result is a sequence of batches where each batch is a function of the input batches. Resulting batches can be computed in parallel and if some computation fails, the framework simply reruns it. Consider an example:



In this example, the framework joins two streams on a sliding window and then the result passes through one more processing stage. The framework considers the incoming streams not as streams, but as set of batches. Each batch has an ID and the framework can fetch it by the ID at any moment of time. So, stream processing can be represented as a bunch of transactions where each transaction takes a group of input batches, transforms them using a processing function, and persists a result. In the figure above, one of such transactions is highlighted in red. If the transaction fails, the framework simply reruns it. It is important that transactions can be executed in parallel.

This simple but powerful paradigm enables centralized transaction management and inherently provides exactly-once message processing semantics. It is worth noting that this technique can be used both for batch processing and for stream processing because it treats the input data as a set of batches regardless to their streaming of static nature.

State Checkpointing

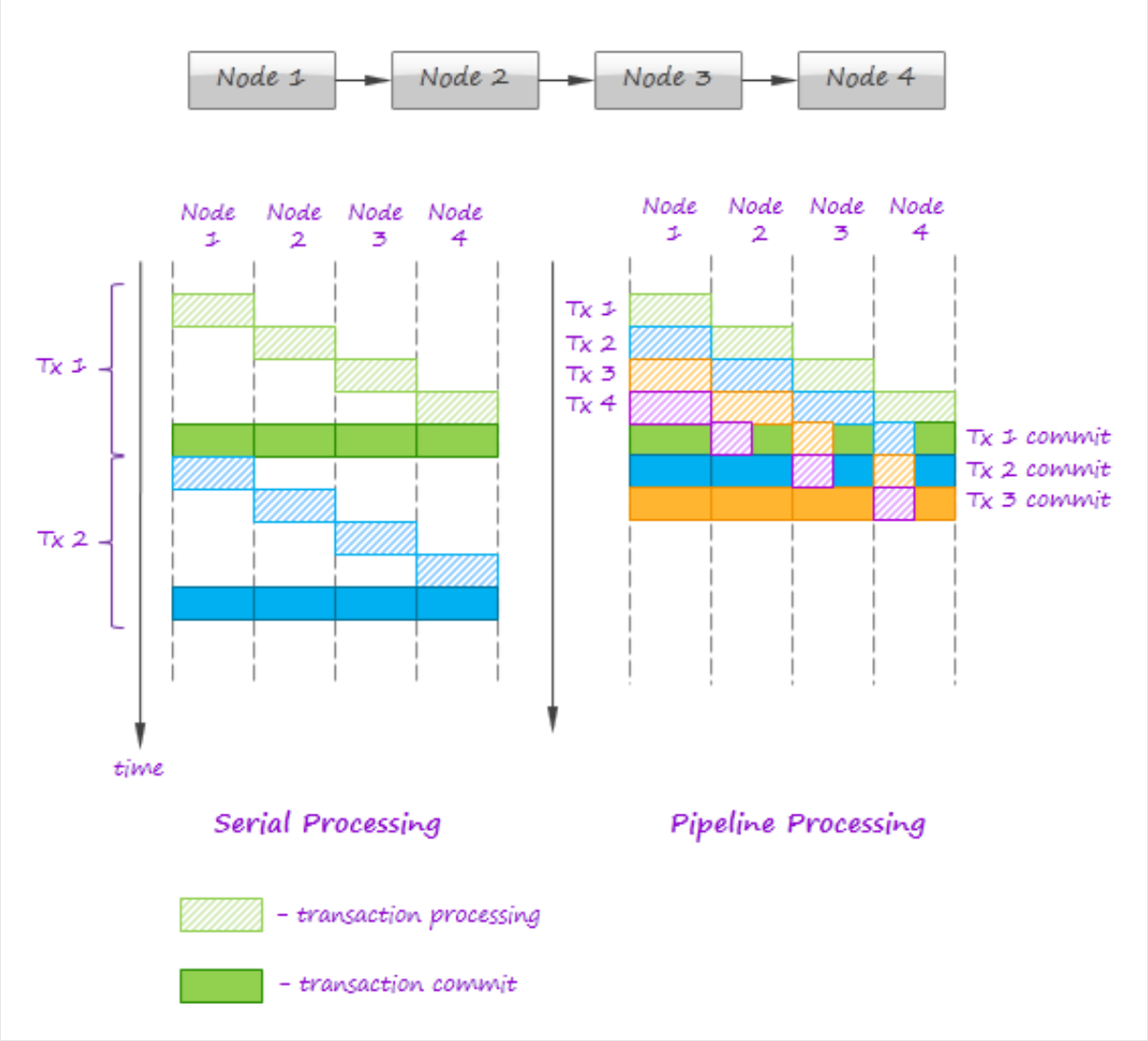
In the previous section we have considered the lineage tracking algorithm that uses signatures (checksums) to provide at-least-one message delivery semantics. This technique improves reliability of the system, but it leaves at least two major open questions:

- In many cases, exactly-once processing semantics is required. For example, the pipeline that counts events can produce incorrect results if some messages will be delivered twice.
- Nodes in the pipeline can have a computational state that is updated as the messages processed. This state can be lost in case of node failure, so it is necessary to persist or replicate it.

Twitter's Storm addresses these issues by using the following protocol:

- Events are grouped into batches and each batch is associated with a transaction ID. A transaction ID is a monotonically growing numerical value (e.g. the first batch has ID 1, the second ID 2, and so on). If the pipeline fails to process a batch, this batch is re-emitted with the same transaction ID.
- First, the framework announces to the nodes in the pipeline that a new transaction attempt is started. Second, the framework to sends the batch through the pipeline. Finally, the framework announces that transaction attempt if completed and all nodes can commit their state e.g. update it in the external database.
- The framework guarantees that commit phases are globally ordered across all transactions i.e. the transaction 2 can never be committed before the transaction 1. This guarantee enables processing nodes to use following logic of persistent state updates:
    - The latest transaction ID is persisted along with the state.
    - If the framework requests to commit the current transaction with the ID that differs from the ID value persisted in the database, the state can be updated e.g. a counter in the database can be incremented. Assuming a strong ordering of transactions, such update will happen exactly one for each batch.
    - If the current transaction ID equals to the value persisted in the storage, the node skips the commit because this is a batch replay. The node must have processed the batch earlier and updated the state accordingly, but the transaction failed due to an error somewhere else in the pipeline.
    - Strong order of commits is important to achieve exactly-once processing semantics. However, strictly sequential processing of transactions is not

feasible because first nodes in the pipeline will often be idle waiting until processing on the downstream nodes is completed. This issues can be alleviated by allowing parallel processing of transactions but serialization of commit steps only, as it shown in the figure below:



This technique allows one to achieve exactly-once processing semantics assuming that data sources are fault-tolerant and can be replayed. However, persistent state updates can cause serious performance degradation even if large batches are used. By this reason, the intermediate computational state should be minimized or avoided whenever possible.

As a footnote, it is worth mentioning that state writing can be implemented in different ways. The most straightforward approach is to dump in-memory state to the persistent store as part of the transaction commit process. This does not work well for large states (sliding windows an so on). An alternative is to write a kind of transaction log i.e. a sequence of operations that transform the old state into the new one (for a sliding window it can be a set of added and evicted events). This approach complicates crash recovery because

the state has to be reconstructed from the log, but can provide performance benefits in a variety of cases.

Additive State and Sketches

Additivity of intermediate and final computational results is an important property that drastically simplifies design, implementation, maintenance, and recovery of in-stream data processing systems. Additivity means that the computational result for a larger time range or a larger data partition can be calculated as a combination of results for smaller time ranges or smaller partitions. For example, a daily number of page views can be calculated as a sum of hourly numbers of page views. Additive state allows one to split processing of a stream into processing of batches that can be computed and re-computed independently and, as we discussed in the previous sections, this helps to simplify lineage tracking and reduce complexity of state maintenance.

It is not always trivial to achieve additivity:

- In many cases, additivity is indeed trivial. For example, simple counters are additive.
- In some cases, it is possible to achieve additivity by storing a small amount of additional information. For example, consider a system that calculates average purchase value in the internet shop for each hour. Daily average cannot be obtained from 24 hourly average values. However, the system can easily store a number of transactions along with each hourly average and it is enough to calculate the daily average value.
- In many cases, it is very difficult or impossible to achieve additivity. For example, consider a system that counts unique visitors on some internet site. If 100 unique users visited the site yesterday and 100 unique user visited the site today, the total number of unique user for two days can be from 100 to 200 depends on how many users visited the site both yesterday and today. One have to maintain lists of user IDs to achieve additivity through intersection/union of the ID lists. Size and processing complexity for these lists can be comparable to the size and processing complexity of the raw data.

Sketches is a very efficient way to transform non-additive values into additive. In the previous example, lists of ID can be replaced by compact additive statistical counters. These counters provide approximations instead of precise result, but it is acceptable for many practical applications. Sketches are very popular in certain areas like internet advertising and can be considered as an independent pattern of in-stream processing. A thorough

overview of the sketching techniques can be found in [5].

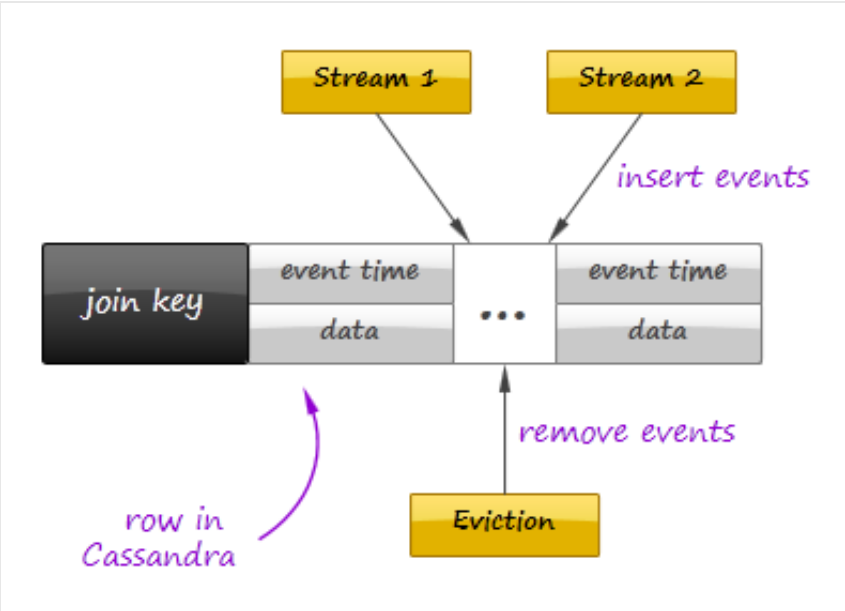
Logical Time Tracking

It is very common for in-stream computations to depend on time: aggregations and joins are often performed on sliding time windows; processing logic often depends on a time interval between events and so on. Obviously, the in-stream processing system should have a notion of application's view of time, instead of CPU wall-clock. However, proper time tracking is not trivial because data streams and particular events can be replayed in case of failures. It is often a good idea to have a notion of global logical time that can be implemented as follows:

- All events should be marked with a timestamp generated by the original application.
- Each processor in a pipeline tracks the maximal timestamp it has seen in a stream and updates a global persistent clock by this timestamp if the global clock is behind. All other processors synchronize their time with the global clock.
- Global clock can be reset in case of data replay.

Aggregation in a Persistent Store

We already have discussed that persistent store can be used for state checkpointing. However, it not the only way to employ an external store for in-stream processing. Let us consider an example that employs Cassandra to join multiple data streams over a time window. Instead of maintaining in-memory event buffers, one can simply save all incoming events from all data streams to Casandra using a join key as row key, as it shown in the figure below:



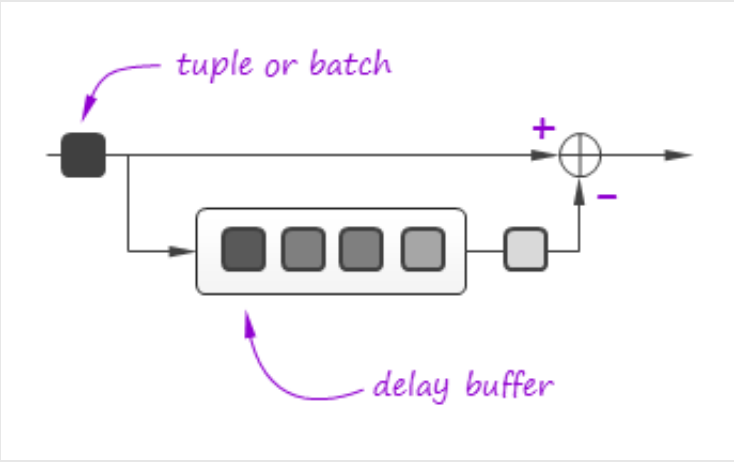
On the other side, the second process traverses the records periodically, assembles and emits joined events, and evicts the events that fell out of the time window. Cassandra even can facilitate this activity by sorting events according to their timestamps.

It is important to understand that such techniques can defeat the whole purpose of in-stream data processing if implemented incorrectly – writing individual events to the data store can introduce a serious performance bottleneck even for fast stores like Cassandra or Redis. On the other hand, this approach provides perfect persistence of the computational state and different performance optimizations – say, batch writes – can help to achieve acceptable performance in many use cases.

Aggregation on a Sliding Window

In-stream data processing frequently deals with queries like "What is the sum of the values in the stream over last 10 minutes?" i.e. with continuous queries on a sliding time window. A straightforward approach to processing of such queries is to compute the aggregation function like sum for each instance of the time window independently. It is clear that this approach is not optimal because of the high similarity between two sequential instances of the time window. If the window at the time T contains samples {s(0), s(1), s(2), ..., s(T-1), s(T)}, then the window at the time T+1 contains samples {s(1), s(2), s(3), ..., s(T), s(T+1)}. This observation suggests that incremental processing might be used.

Incremental computations over sliding windows is a group of techniques that are widely used in digital signal processing, in both software and hardware. A typical example is a computation of the sum function. If the sum over the current time window is known, then the sum over the next time window can be computed by adding a new sample and subtracting the eldest sample in the window:

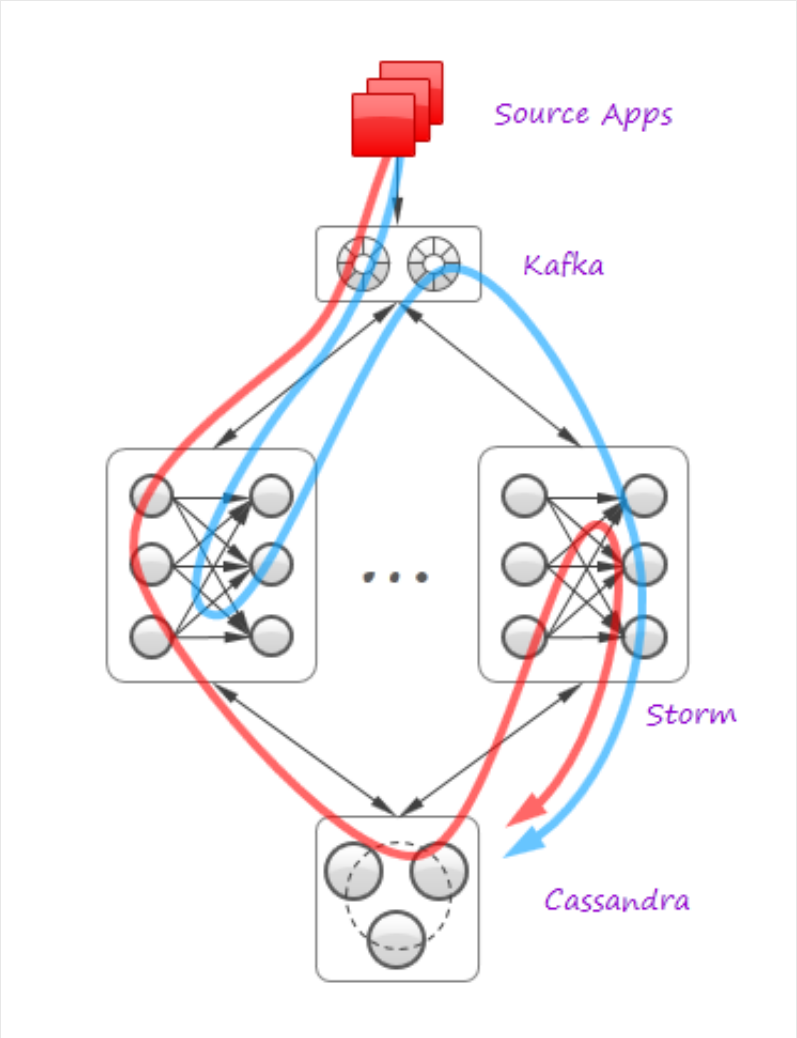


Similar techniques exist not only for simple aggregations like sums or products, but also for

more complex transformations. For example, the SDFT (Sliding Discreet Fourier Transform) algorithm [4] is a computationally efficient alternative to per-window calculation of the FFT (Fast Fourier Transform) algorithm.

Query Processing Pipeline: Storm, Cassandra, Kafka

Now let us return to the practical problem that was stated in the beginning of this article. We have designed and implemented our in-stream data processing system on top of Storm, Kafka, and Cassandra adopting the techniques described earlier in this article. Here we provide just a very brief overview of the solution – a detailed description of all implementation pitfalls and tricks is too large and probably requires a separate article.



The system naturally uses Kafka 0.8 as a partitioned fault-tolerant event buffer to enable stream replay and improve system extensibility by easy addition of new event producers and consumers. Kafka's ability to rewind read pointers also enables random access to the incoming batches and, consequently, Spark-style lineage tracking. It is also possible to point the system input to HDFS to process the historical data.

Cassandra is employed for state checkpointing and in-store aggregation, as described earlier. In many use cases, it also stores the final results.

Twitter's Storm is a backbone of the system. All active query processing is performed in Storm's topologies that interact with Kafka and Cassandra. Some data flows are simple and straightforward: the data arrives to Kafka; Storm reads and processes it and persist the results to Cassandra or other destination. Other flows are more sophisticated: one Storm topology can pass the data to another topology via Kafka or Cassandra. Two examples of such flows are shown in the figure above (red and blue curved arrows).

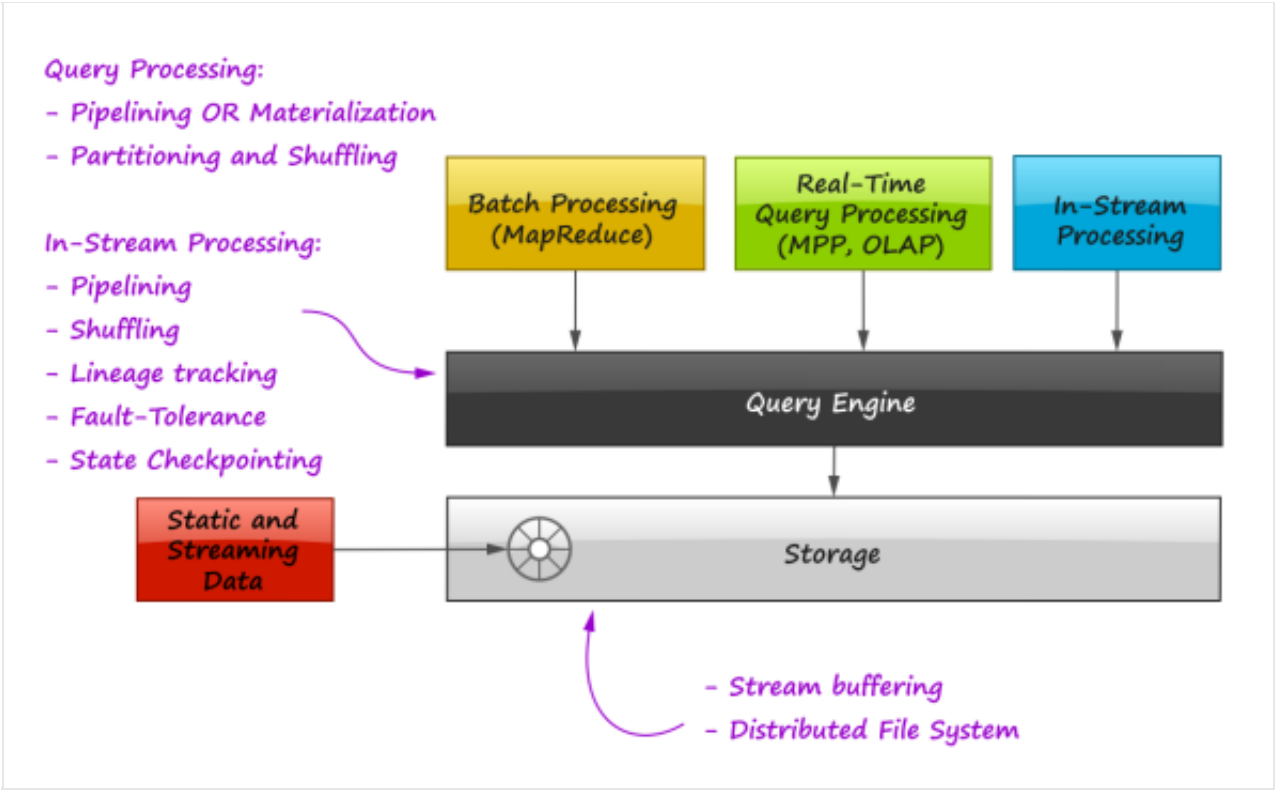
Towards Unified Big Data Processing

It is great that the existing technologies like Hive, Storm, and Impala enable us to crunch Big Data using both batch processing for complex analytics and machine learning, and real-time query processing for online analytics, and in-stream processing for continuous querying. Moreover, techniques like Lambda Architecture [6, 7] were developed and adopted to combine these solutions efficiently. This brings us to the question of how all these technologies and approaches could converge to a solid solution in the future. In this section, we discuss the striking similarity between distributed relational query processing, batch processing, and in-stream query processing to figure out the technologies that could cover all these use cases and, consequently, have the highest potential in this area.

The key observation is that relational query processing, MapReduce, and in-stream processing could be implemented using exactly the same concepts and techniques like shuffling and pipelining. At the same time:

- In-stream processing could require strict data delivery guarantees and persistence of the intermediate state. These properties are not crucial for batch processing where computations can be easily restarted.
- In-stream processing is inseparable from pipelining. For batch processing, pipelining is not so crucial and even inapplicable in certain cases. Systems like Apache Hive are based on staged MapReduce with materialization of the intermediate state and do not take full advantage of pipelining.

The two statement above imply that tunable persistence (in-memory message passing versus on-disk materialization) and reliability are the distinctive features of the imaginary query engine that provides a set of processing primitives and interfaces to the high-level frameworks:



Among the emerging technologies, the following two are especially notable in the context of this discussion:

- Apache Tez [8], a part of the Stinger Initiative [9]. Apache Tez is designed to succeed the MapReduce framework introducing a set of fine-grained query processing primitives. The goal is to enable frameworks like Apache Pig and Apache Hive to decompose their queries and scripts into efficient query processing pipelines instead of sequences of MapReduce jobs that are generally slow due to materialization of intermediate results.
- Apache Spark [10]. This project is probably the most advanced and promising technology for unified Big Data processing that already includes a batch processing framework, SQL query engine, and a stream processing framework.

References

1. A. Wilschut and P. Apers, “Dataflow Query Execution in a Parallel Main-Memory Environment “
2. T. Urhan and M. Franklin, “XJoin: A Reactively-Scheduled Pipelined Join Operator“
3. M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, “Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters”
4. E. Jacobsen and R. Lyons, “The Sliding DFT“
5. A. Elmagarmid, Data Streams Models and Algorithms

6. N. Marz, “Big Data Lambda Architecture”
7. J. Kinley, “The Lambda architecture: principles for architecting realtime Big Data systems”
8. http://hortonworks.com/hadoop/tez/
9. http://hortonworks.com/stinger/
10. http://spark-project.org/