# Report: Stream and Batch Processing Systems for Big Data

Johannes Jestram
*DBSem summer term 2019*
*T*U-Berlin
jestram@posteo.de

***Abstract*—This report is part of our work on the module "Database Seminar: Foundations of Database Systems". It will provide the reader with an overview on how processing big data is a difficult task for traditional database systems. We will focus on the system architecture of stream processing engine Aurora and batch processing engine Stratosphere, which were developed for coping with different challenges regarding big data.**

***Index Terms*—stream, batch, big data, Aurora, Stratosphere**

## I. Introduction

In recent years, mobile devices, the digitalization of workplaces and private life, and software applications themselves have rapidly increased the amount of produced data. The amount of data has grown in several dimensions. First, the volume of data increased. Second, the variety of data has grown, e.g. there are lots of unstructured textual data created on the internet that require further processing. Third, the velocity of the produced data has sped up, as there is a growing number of devices each producing increasing volumes of data.

This development has led to new use cases for data, e.g. by enabling the use of neural networks that require lots of training data to function properly. Furthermore, whole new business models have developed solely around the usage of large amounts of data. One prominent example is the case of Cambridge Analytica [1], [2], which analyzed user data on social networks to enable highly precise targeting of different demographic groups via social networks, with their working results going as far as interfering in democratic elections. Those new use cases and business models have added much value to data. But with the ongoing automization of business- as well as private tasks, the question arises, if and how all the new data can be trusted. That concern is strengthened by bots on social networks, the increased use of artificial intelligence and the fact that technology might simply fail at times and hence produce invalid data. The problem of the veracity of data is an important issue, which must be handled when working with large amounts of data.

The keywords mentioned before, i.e. volume, variety, velocity, value and veracity, define the catchword big data.

From a technical point of view, the question of how to actually process big data arises. Traditionally, relational database management systems (RDBMSs) first come to mind. While the RDMBSs are able to process large volumes of data, i.e. by parallelizing the workload, they are not a great fit. The reason is that they are not able to efficiently scale-out with increasing load, as they must replicate and restructure their databases. Additionally, because they were created for processing structured data, RDBMSs struggle with semi-structured and unstructured data. If the processed data has real time requirements, RDBMSs also suffer of a suboptimal processing speed. Beside relational systems, there are alternative database management systems which e.g. specialize in efficiently saving and querying semi- and unstructured data, often referred to as NoSQL. But those systems face similar problems regarding dynamic scale-out and general efficiency in processing. When it comes to performance, NoSQL systems might even perform worse than RDBMSs [3]. A newer answer to big data and the problems of RDBMS is MapReduce [4], which also does not necessarily show better performance than RDBMSs [5].

There are two prominent use cases for big data. The first is treating data that arrive as a continuous stream. Those data may have real time requirements and be produced by multiple sources in a heterogenous way. One example is a position tracking system aiming to locate cars and detect traffic jams, while additionally providing alternative routes in the case of a traffic jam. The second use case is the need for analyzing data of large volume, a so called batch. Such batches, with sizes ranging from several terabytes to petabytes, might be provided by relational databases, data warehouses, different devices or a combination of the aforementioned. The challenge here is to provide valid results in a reasonable time while receiving unstructured and heterogenous data. For both use cases, there is an obvious need for systems that scale, are intuitive to use, and are able to provide valid results. Traditional database systems are not a good fit for such tasks. This is the reason that many new systems have been developed around the world.

In this report we will present two prominent big data processing engines for different use cases. First (Section II), we will introduce Aurora [6], a streaming engine for monitoring applications, which was developed in 2003. The second topic (Section III) will be Stratosphere [7], a batch processing platform for analytical tasks published in 2014 which has since evolved into Apache Flink [8]. After explaining the need for such systems (Sections II-A, III-A), we will describe both systems in detail. Furthermore, we will discuss possible weaknesses of each system. We will also discuss the development of both projects (Section IV) and extend the reader's view by briefly presenting an alternative way of handling data streams (Section V). The report is concluded (Section VI) by

summarizing the presented results and providing an outlook on current research topics and possible future developments.

## II. Stream Processing with Aurora

Aurora is a stream processing engine for monitoring applications and was developed by members of Brandeis University, Brown University and Massachusetts Institute of Technology. It has been published as open source software in 2003 and marked the first significant software publication in a continuing research effort on stream processing. Because stream processing has been a task of increasing importance, the authors developed a new and specialized engine. Following up, we will explain the reasons why RDBMSs are no sufficient solution for handling data streams (Section II-A) and discuss Aurora as an alternative (Section II-B - II-F).

### A. Stream Processing using Traditional RDBMSs

Most streaming applications have two critical properties in common. First, they must be able to handle predefined events. For example, a temperature monitor for a chemical plant must perform an alert, when the temperature passes a certain threshold. Second, the results of the processed data queries are expected in real time. A temperature alert that is activated only after minutes of processing is useless.

Traditional relational databases are read centered and optimized for potentially huge sized databases, while maintaining a consistent state and staying online. We will now discuss the main assumptions, around which database management systems are built, and what are the reasons for their poor performance regarding stream processing.

The first assumption is that traditional databases only start working, when there is some command scheduled by the user. This principle is called *human active, database passive* (HADP). A data stream management system (DSMS), on the other hand, works vice versa: it has pre-implemented queries by the user, and processes every arriving tuple of data automatically. It is a *database active, human passive* approach.

Second, database management systems were designed around the thought that the current state of the data is the only one that matters. Of course, some aggregated historical data might be archived in a data warehouse, but that is not a traditional database anymore. When working with streams, the user is interested in the history of data. In the aforementioned chemical plant, a supervisor has a natural interest in knowing how the temperatures of the steam cracker developed over time, e.g. for maintenance reasons or to recognize harmful trends in the temperature-over-time-graph.

Furthermore, the concept of triggers evolved after RDBMSs were first introduced. For that reason, triggers were added as an afterthought. This has led to a suboptimal performance of databases with many triggers. Also, there are possible inconsistencies when nesting triggers. On the contrary, the usage of data streams relies heavily on triggers, hence they should be treated not as second-, but as first-class-citizens.

When working with relational databases, the user can expect correct answers to queries on the database. The ACID principle, depending on how firm it is implemented in a database, implies correct data and by that correct query results [9]. On a data stream, query results cannot be guaranteed to be correct at all times. Data might be lost during physical transfer, or dropped, because of a system overload. Also, some operations, such as aggregations, change results when performed on different times on a virtually infinite data stream.

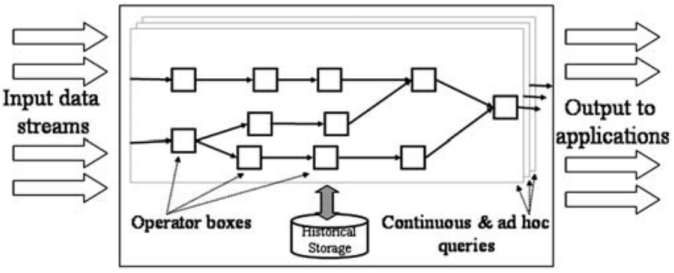


Fig. 1. Aurora's system model [6].

Finally, while there is a need for efficient processing, queries on a traditional RDBMSs do not need to be performed in real time. Streaming applications, on the other hand, rely heavily on the ability to react in real time to changes in the input data.

With some workarounds and careful tweaking of relational database management systems, they are able to perform rudimentary stream processing tasks. But as we explained, they are a suboptimal fit, regarding configuration effort as well as performance [5]. The differences in the underlying assumptions that we described are the reason the developers of Aurora saw the need for a fundamentally different approach to data streaming engines.

### B. Aurora System Overview

The Aurora system was designed following the assumptions for DSMS. As seen in Fig. 1, the system model follows a boxes-and-arrows approach. After data from multiple sources have arrived at the system, different operators modify the data to generate one or more outputs. The operators are connected to each other by the user, thus providing the ability to form complex queries. One operator may have multiple outputs. Additionally, a historical storage might be used for persisting interesting information or for setting recovery points. Historical storage is organized as a B-tree. Users can set up and manage the system with a graphical user interface. Input data sources can be computer programs or sensors. The data arrive in form of tuples, with one field of each tuple serving as a unique identifier.

To satisfy different needs to access the data stream, Aurora provides three different types of queries. The user is (1) able to define continuous queries. Furthermore, the system (2) supports views that might include historical data. Finally, users can perform (3) ad-hoc queries to receive information not included in the continuous queries.

Fig. 2 shows Aurora's query model. Each box represents an operator, the connecting arrows represent the data flow. There are predefined connection points, represented by the dots, i.e. before $b_1$, and between $b_1$ and $b_2$. Each connection point

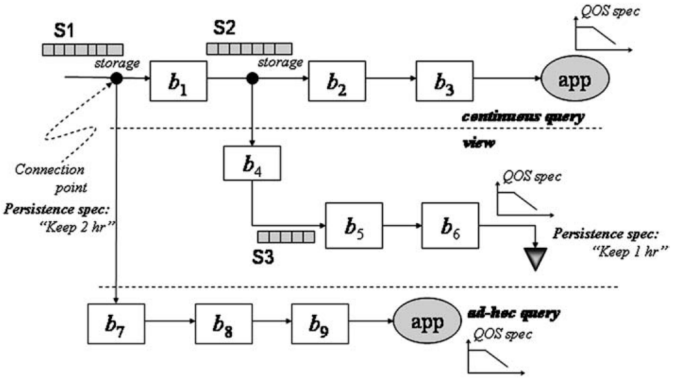


Fig. 2. Aurora's query model [6].

possesses a dedicated storage (*S1, S2*), and ad-hoc queries might be connected only to a connection point. For every connection point's dedicated storage the time of persistence can be configured individually. This storage enables ad-hoc queries and views on historical data. One operator may have multiple outputs. In such a case, a subnetwork is created for each output. Further subnetworks are created, when new queries are connected to connection points. Also note that each operator has its own output buffer, which can be accessed by the succeeding operators.

### C. Optimization in Aurora

Optimizers of RDBMSs aim to minimize the number of iterations over large data sets. When handling streams, a system needs to process data as it arrives. The amount of computation per arriving tuple is small, but the optimizer expects a high number of operations. Also, the data arrival rates might be enormous. Aurora's optimizer focuses on two areas.

The first area is the optimization of continuous queries. The system gathers runtime statistics, such as cost of box execution and box selectivity, i.e. the relative number of output tuples per input tuple. Three optimization tactics are employed. First, Aurora aims to reduce the size of each tuple, by using projections as early as possible. This way, the system eliminates unused operators. Second, combining operators might improve execution speed by getting rid of box-execution overhead. Map and filter operations may be combined with almost all other operators (more on the different operators in Section II-E). Last, if two operators commute, they can be safely reordered. The optimizer reorders two operators, if the resulting amount of computation is smaller than before reordering. We define two boxes $b_i$ and its successor $b_j$. Each operator's execution cost is $c(b_i)$ and $c(b_j)$ and their selectivities $s(b_i)$ and $s(b_j)$. The optimizer reorders the operators, if (1) is fulfilled:

$$c(b_\mathrm{i}) + c(b_\mathrm{j}) * s(b_\mathrm{i}) > c(b_\mathrm{j}) + c(b_\mathrm{i}) * s(b_\mathrm{j}) \qquad (1)$$

To make use of the aforementioned optimization tactics, the optimizers periodically performs the following algorithm.

1) Select subnetwork to optimize.

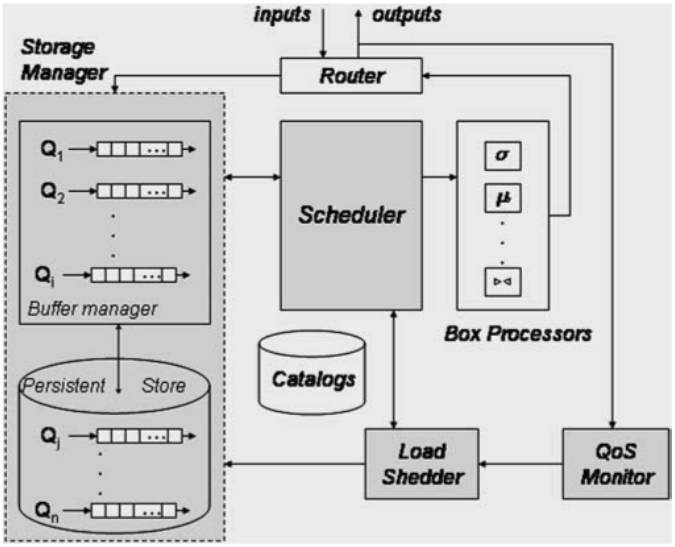


Fig. 3. Overview on Aurora's runtime components [6].

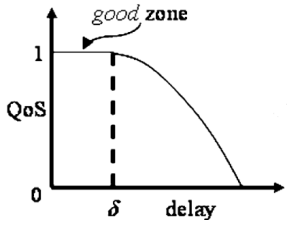


Fig. 4. Quality of Service in relation to the input-output delay [6].

2) Find all connection points surrounding the subnetwork.
3) Hold all input messages at connection point before the subnetwork (upstream).
4) Drain all messages out of the subnetwork through connection points following the subnetwork (=ownstream).
5) Optimize by moving selections, combining boxes and reordering operators. This creates a new subnetwork.
6) Connect new network to the connection points and route the data through the network.
7) Repeat periodically.

This algorithm makes sure that no tuples are lost and that the system can resume its operation seamlessly.

When the user schedules an ad-hoc query, the second area of the optimizer is utilized. If the query accesses historical data, the optimizer is able to perform indexed lookups on the B-tree of the historical storage. This works for filter and join operations and only, if the B-tree is indexed on the fields corresponding to the user's query.

### D. Runtime Operation in Aurora

Fig. 3 shows the different components Aurora employs at runtime. We will focus on the Quality of Service Monitor, Storage Manager and Scheduler.

For evaluating the performance of the system, Aurora employs a Quality of Service (QoS) Monitor. The user may define multiple functions, which specify the decline of the

QoS in relation to different, user defined, metrics. At least one function, i.e. showing the decline of QoS in relation to the time delay between a tuple arriving and being put out of the system, must be implemented. Fig. 4 shows the mandatory function. Extending this, other functions, such as the percentage of delivered tuples ore more complex metrics, such as QoS for each value of the output, meaning a change of the abscissa metric, might be added. The QoS Monitor continuously gathers runtime statistics to evaluate the QoS by using the user defined functions as performance indicators.

Because disk I/O operations are expensive, the Aurora Storage Manager's (ASM) task is to minimize the amount disk I/O. The ASM works in two areas. First, it performs queue management. As stated above, each operator has a queue at its output, which is shared by all directly succeeding operators. Each succeeding box maintains two pointers on its preceding queue. One is the head, which marks the oldest tuple in the queue not yet processed by the operator. The other is the tail, which marks the oldest tuple that is not needed by the operator anymore. Head and tail define the window of tuples that an individual operator needs from the queue. Older values might be discarded. The queues are organized in 128kB blocks and may be increased or decreased block-wise in times of overflow or underflow. The ASM has a buffer pool in main memory, where it can load blocks of queues.

To keep track of which queue-blocks to keep in main memory the developers created a tabular data structure that is shared by the Scheduler and ASM. Each row represents one operator box and has different attributes, i.e. scheduling priority (maintained by the Scheduler), percentage of its queue currently in main memory (maintained by ASM) and a flag indicating if the box is running. When the ASM discovers a block of a queue in main memory that is not currently running, it loads a block of a queue with higher priority instead. Also, when main memory space is needed for a block from disk, the block belonging to the lowest priority queue gets replaced. This ASM-workflow results in top prioritized boxes most often being already in main memory when needed.

Another task of the ASM is connection point management. To be more efficient, ASM performs insertions and deletions to the historical storage in batches.

In addition to storage management, the scheduling of operations is of importance. There are several issues that the Scheduler must address to achieve good overall performance. The potentially large scale of the system and real-time requirements describe the frame, in which the Scheduler operates. Also, there might be dependencies between box executions that the Scheduler needs to keep track of. Finally, the system can not schedule solely on QoS specifications, because that might lead to exploding end-to-end tuple processing costs. The number of box calls per tuple as well as I/O operations should be minimized. To meet these requirements, the Scheduler uses two tactics. First, it generates *tuple trains*, which means that it queues as many tuples as possible before processing them and processes each train as a whole. Second, the trains are processed by as many consecutive boxes as possible at once.

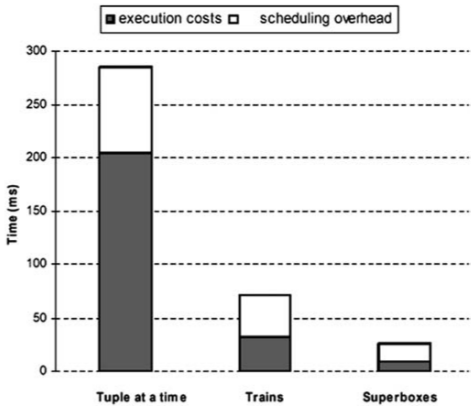


Fig. 5. Influence of scheduling tactics on end-to-end tuple processing time [6].

This tactic is called *superbox scheduling*. Both tactics work exceptionally well and increase end-to-end processing time drastically. Execution costs as weill as scheduling overhead are reduced, as shown in Fig. 5.

If there is a system overload, the Load Shedder is able to use QoS information and employ two tactics to increase performance. It is able to drop tuples and also filter tuples. The problem with dropping tuples is that there are more important and less important tuples (which may or may not be specified by a user created QoS function). This issue is addressed by the tactic of filtering tuples, which tries to use QoS information to drop less important tuples.

### *E. Stream Query Algebra*

To define queries for Aurora's operator boxes, the developers have created a new declarative query language, called Stream Query Algebra (SQuAl). It consists of eight explicit operators, which are similar to other established query languages. The reason for creating a new query language has been that data streams have some characteristics, because of which traditional data manipulation languages are not applicable. For example, aggregating operators work different, when they are executed on possibly infinite streams. SQuAl operators are classified into order agnostic and order sensitive ones.

There are three order agnostic operators in SQuAl: *map*, *filter* and *union*. *Map* serves as a generalized projection and is able to manipulate each attribute of an incoming tuple. Hence, it is able to change the schema of a processed tuple. *Filter*, on the other hand, does not change the schema of a tuple, but applies user defined predicates to different attributes. It is able to route tuples to different output streams based on a predicate, which means that filter can serve as a fan-out operator. On the other hand, *union* combines two or more streams with a common schema into one stream, therefore serving as a fan-in operator. The tuples themselves are not changed by *union*.

The order sensitive operators are *Bsort*, *aggregate*, *join* and *resample*. Because of their nature performing operations on a set of tuples it is important to specify an *order* and a window in which tuples must arrive. To satisfy this need, the user must specify an order function $O$(On $A$, Slack $n$, GroupBy

$B_1$, , $B_i$) for each operation in addition to the basic function of the operation itself. *A* is the attribute in which the order is assumed, *n* defines a possible slack (default is $n = 0$) and $B_{a,,i}$ are attributes on which the input is grouped by. With a nonzero slack, the restriction on the assumed arrival order can be relaxed. With a slack of $n = 2$, a window of *size 3* is defined for the respective operator. Considering the *Bsort* operator, which orders input tuples of a stream, the slack defines the buffer size for ordering tuples. With a *slack of 2*, the buffer has three fields on which *Bsort* performs the ordering. This means that out of order tuples, which arrive up to two tuples late, can be put into order. With *aggregate*, a user defined aggregation of several values is possible. To take into account that there is an infinite stream of tuples *aggregate* needs a specified window, after which it performs the aggregation. It also needs a step function, which defines how the window advances. The *Join* operator joins tuples of two streams together on a user defined predicate *P* and a defined window *size* for an attribute. For example, the user could join two streams of position data of two cars, that were in the same area (*P*) within a given time (*Size*). The *resample* operator performs an interpolation for each value on a stream $S_1$ based on values in a stream $S_2$. As *join* and *aggregate*, *resample* also needs a window, on which it operates. Last, there is an implicit *split* operation that is performed by simply connecting a box not to one, but to several other boxes, thus splitting the output into several disjunct streams.

### F. Areas for Improvement in Aurora

Aurora presented a new approach to handling data streams. Many ideas were new, and the performance of the system was excellent compared to RDBMSs. But nevertheless, the developers identified two fields for further improvement. First, Aurora does neither support parallel, nor distributed execution. It lacks the ability to dynamically scale out in times of heavy load. Another problem with running only on one system is that the whole system shuts down when the computer has a problem. This issue was addressed in further research. *Medusa* was developed as a distributed execution engine for aurora. Both projects were combined to form *Borealis* [10], a distributed stream processing engine. The project was commercialized in 2003 as StreamBase Inc.

Another inconvenience is that Aurora's operator set is not extensible. In an enterprise environment, being able to add operators for specific business needs might be of interest.

Furthermore, the developers aimed to address the ability of Aurora "to cope with missing and imprecise data values, which are common in applications involving sensor-generated data streams" [6] in further research.

## III. Batch processing with Stratosphere

The Stratosphere project was a research project by institutes of Technical University Berlin, Humboldt University Berlin and Hasso Plattner Institute Potsdam. It was founded in 2009 and continued with publications until 2016 [11]. After research on parts of the system had advanced far enough, the combined system was introduced as "The Stratosphere platform for big data analytics" in 2014 [7]. The goal of the project was "to jointly research and build a large-scale data processor based on concepts of robust and adaptive execution" [12] and for the user to be able to comfortably execute analytical tasks on large batches of data [7]. Also in 2014, the project evolved into a Apache top level project under the name of Apache Flink [11]. As for Aurora, we will now referring to [7] - describe the reasons for developing a system like Stratosphere and afterwards explain its main components.

### A. Problems of batch processing with traditional RDBMSs

Similar to stream processing, traditional (parallel) RDBMSs are able to perform analytical tasks on large batches of data, but again they are slow and inefficient at this task. In addition, RDBMSs have difficulties in handling semi-structured and unstructured data, which is a issue, because many analytical tasks are performed on such data. Finally, the authors add, there is little comfort from the user's point of view in using RDBMSs, thus making the work inefficient.

### B. Stratosphere system overview

To address the issues of processing large amounts of data at once and in a reasonable time, Stratosphere offers several features. First, it performs in-situ data processing, thereby minimizing memory usage. Second, Stratosphere offers a declarative query language. The authors think that the high level of abstraction, which declarative languages offer, makes users more productive. Furthermore, the system automizes program parallelization and optimization and has a scalable, distributed execution engine. User defined functions are treated as first class citizens and the set of operators is extensible. Last, Stratosphere supports iterative programs by default. Because of those features, Stratosphere is useful in many use cases, such as data warehousing, information extraction, data cleansing, graph analysis and statistical analysis.

The system consists of several layers, as seen in Fig. 6. Queries are written in Meteor, Stratosphere's declarative query language. A Meteor script is translated into a directed, acyclic graph (DAG) by the Sopremo layer. PACT then further translates the Sopremo DAG by braking Sopremo operators down to PACT operators, which consist of five pre-implemented second order functions combined with user defined functions (UDFs). Afterwards, the PACT DAG gets optimized in four steps, resulting in a Nephele Job Graph. Nephele is Stratosphere's parallel execution engine and takes care of scheduling tasks on different worker nodes. The system can connect to several popular file systems, including YARN [1], Amazon EC2 [2], and Hadoop Distributed File System [3]. Note that queries can be either input as Meteor scripts, Sopremo graphs or PACT programs. That allows different levels of abstraction regarding queries.

[1] https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[2] https://aws.amazon.com/de/ec2/

[3] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

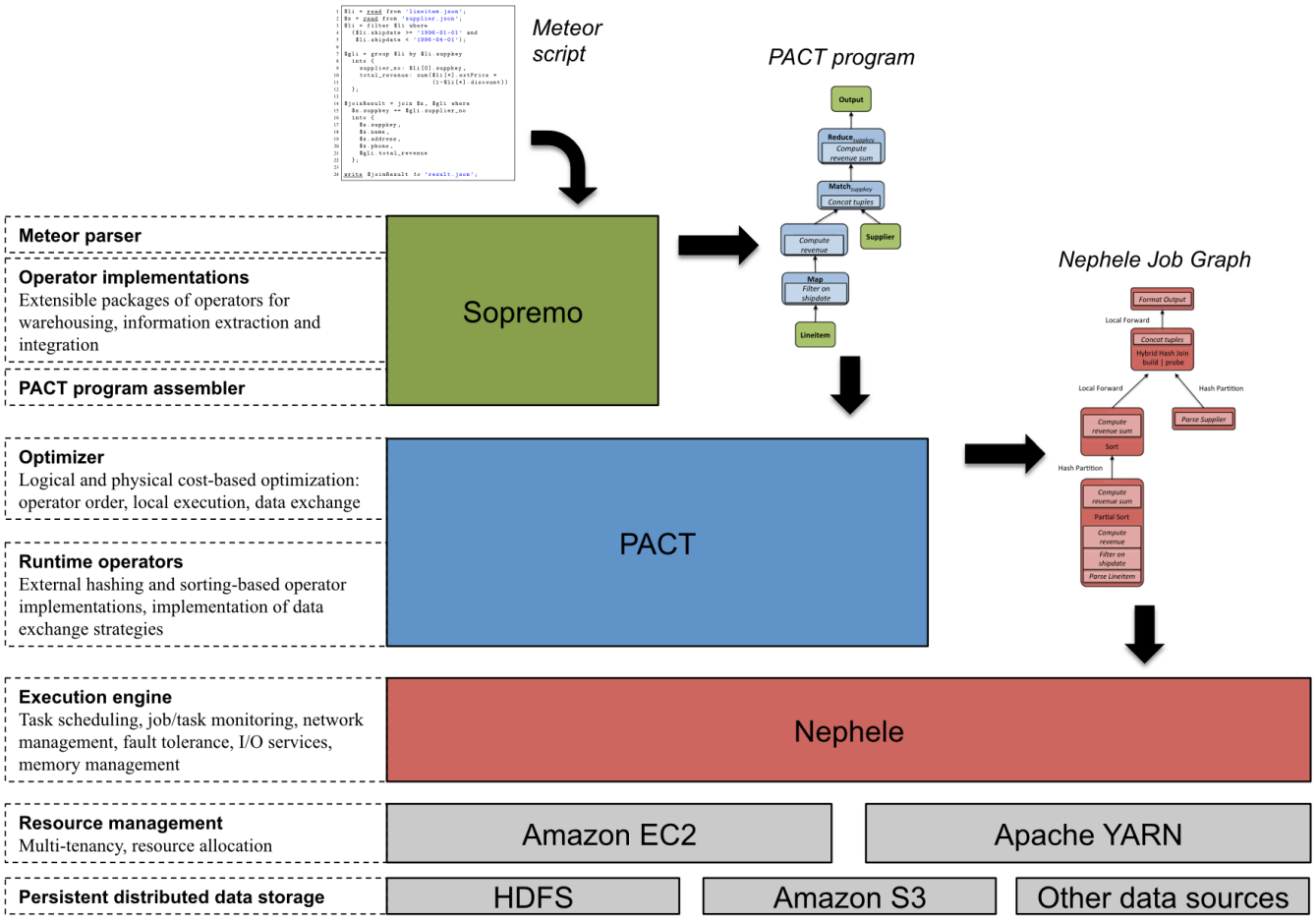


Fig. 6. Stratosphere system overview. Notice the layered system that connects to several different popular file systems [7].

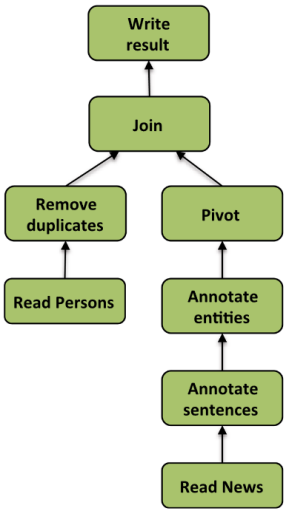


Fig. 7. A directed acyclic graph (DAG) in the Sopremo layer. Sopremo can take a DAG as an input or translate a Meteor script to a DAG [7].

### C. Meteor and Sopremo

Meteor serves as the query language for the topmost Sopremo layer. Thereby, Meteor abstracts queries from a Sopremo graph into a structured query language. This relation implies that every Meteor operator must be backed by a corresponding Sopremo operator. An example Sopremo DAG is shown in Fig. 7. Each node in the DAG resembles a Meteor operation and each Meteor variable is represented by a vertex in the graph. Sopremo operators may be elemental or composed out of elemental operators. As of 2014, Sopremo already supported a wide range of operations for Meteor. A detailed listing of supported operations is shown in [7].

### D. PACT

The PACT layer's job is to provide information about parallelization. PACT stands for "parallelization contract". Each PACT operator, or simply PACT, consists of a second order function and a UDF. Stratosphere started with a set of five PACT second order functions in 2014: *Map, Reduce, Cross, Match, and CoGroup*. The purpose of the second order functions is that each of them forms different *parallelization units* (PU) when processing data. For example, *Reduce* creates a PU for each key of the input. The PUs define the way that incoming data may be distributed to different computers. Because a second order function itself describes only how data is grouped into PUs, each PACT needs a UDF to actually

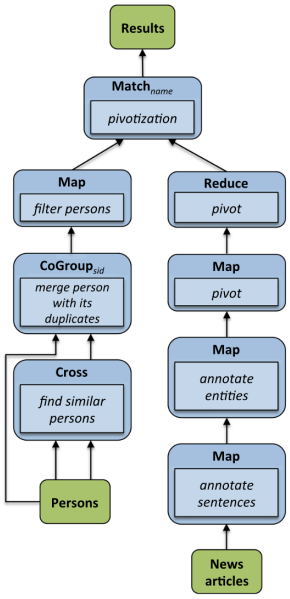


Fig. 8. DAG of a PACT program. Every Sopremo operator has to be backed by a PACT operator or program [7].

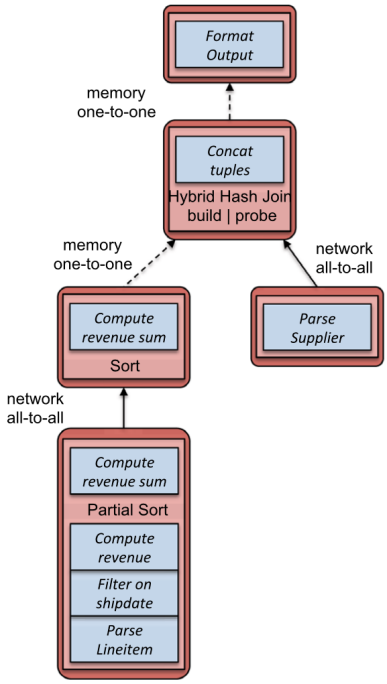


Fig. 9. The Nephele job graph is created after optimization and contains information about how and where to physically execute the query [7].

perform any operation on the data. Each Sopremo operator must be backed by a PACT, consisting of a second order function and an implemented UDF. The set of second order functions, as well as the set of pre-implemented PACTs backing Sopremo operators is extensible. That allows power users to freely extend the operator set. For example, a company could decide to implement a PACT for a complex function, that is often used in the company's working context, and must otherwise be composed out of many Sopremo operators.

Because many data analysis tasks can not be algorithmically solved in one iteration, the PACT programming model supports iterative programs. Such programs can either perform *bulk iterations* or *incremental iterations*. A *bulk iteration* consumes the whole data set in each iteration and a *incremental iteration* has a result set and a working set. Only the working set is consumed by the iteration, modifying the result set after each iteration.

Because it supports more operators than MapReduce and natively embeds iterative operations, the PACT model can be seen as a generalization of MapReduce's operator model.

PACT transforms a Sopremo DAG into a PACT DAG by translating each Sopremo operator into the corresponding PACT operator(s). Fig. 8 shows the result of the translation of the Sopremo graph from Fig. 7.

### *E. Stratosphere's Optimizer*

Before a program gets executed, it is optimized. First, the PACT DAG gets translated to a internal representation. Data sources and sinks, as well as some internal operations are added to the DAG. Afterwards, the optimization itself is performed in two stages. The first stage, logical optimization, consists of the reordering of operators. Because the system does not use relational algebra, traditional rules for reordering, such as the commutative property, do not apply. The UDFs make it difficult to generalize reordering rules. The authors developed two conditions, under which two operators may be switched. First, there may be no conflicting read and write operations between the two respective operators. This is only safely fulfilled, if there are only read operations on the same fields of a data tuple. As soon as one operator writes the same field that the other reads from, they do not commute. Second, the reordering must not change the input group of group-based operators, because this could change the semantics of the operator. To gather the necessary information about read/write accesses and group cardinalities, the optimizer performs static code analysis on the user defined functions. The authors designed a new algorithm, that applies these rules to a given DAG. It decomposes a DAG into subtrees, then optimizes each subtree and finally recomposes a DAG.

After the logical optimization comes the physical optimization. The main goal for physical execution is to minimize network and disk I/O, as these operations determine overall system performance. Different execution strategies, such as repartition, broadcast data transfer strategies and also local execution strategies, are employed. In addition, the optimizer keeps track of interesting properties, from which an operator might benefit. Such physical data properties include sorting, grouping and partitioning. The physical optimization algorithm starts on the data sink and traverses the graph depth-first, while keeping track of interesting properties. When it reaches a data source, it optimizes on its way back to the sink, remembering the interesting properties.

The last optimization step results in the Nephele Job Graph

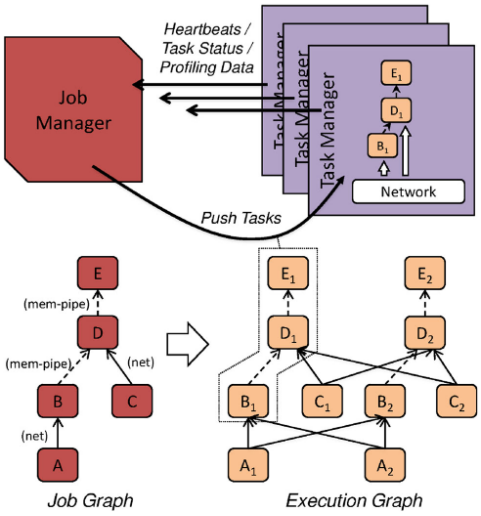


Fig. 10. Nephele's job manager assigns tasks to different task managers. One task might be executed on different workers in parallel [7].

(Fig. 9). Local operations are clustered together to signalize that they form one execution unit. All data forwarding techniques are added to the graph and it is forwarded to Nephele.

### F. Nephele

Nephele is Stratosphere's distributed execution engine. It takes a Nephele Job Graph as input. The system uses a master/worker pattern to distribute tasks and the necessary communication is provided via messaging. Fig. 10 shows that there is one Job Manager that assigns tasks to the different task managers, which represent different nodes in a distributed system. In the beginning, Nephele assigns the jobs containing the data sources to different Task Managers. The following tasks are executed lazily: when a Task Manager has finished its job and asks the Job Manager where to forward their data, the succeeding tasks are started, if they have not already. When sending data via network, Task Managers are able to compress the data to minimize network throughput. This is a tradeoff between computing power to compress and decompress data and network load.

To provide fault tolerance, Nephele works with log-based rollback recovery, similar to other data flow engines, such as Hadoop [4], Spark [5] and Dryad [6]. The main difference in Stratosphere is that operators are non-blocking, which means that tuples get forwarded as soon as they have been processed at the preceding operator. In a case of failure, some tuples might not have been processed yet, while other might have already been forwarded to following operators. Nephele keeps track of tuples already processed by an operator with a duplication detection system similar to TCP. For the rollback, the optimizer is able to identify possible checkpoints for materialization of intermediate results. Sometimes, however, materializing costs are higher than recomputation costs, leading to the checkpoint being ignored.

Stratosphere is implemented in Java. While this, because of the extensive libraries and far evolved language, is a good fit for programming UDFs, some difficulties arise with regard to core system functionalities of Stratosphere. Java does not support explicit memory control and Java-objects tend to have a large memory overhead. Further, the Garbage Collector needs much memory. Those issues are addressed by careful usage of the Java language. For example, the developers decided to use large byte arrays to reduce object overhead and minimize the Garbage Collectors activity by lazily deserializing fields of the byte arrays into objects.

### G. Stratosphere performance

The developers tested Stratosphere against Apache Hadoop's MapReduce [7], Apache Hive [8] and Apache Giraph [9] in different benchmarks, such as TeraSort, Word Count, a relational query, triangle enumeration of a graph, and a connected components algorithm. The results have shown that Stratosphere has comparable or better performance than the competitors. As a reason, the authors have identified execution layer features, such as PACT implementations and the push-based communication between Task Managers.

### H. Areas for improvement in Stratosphere

In 2014, the authors defined some areas for further research. They wanted to add support for a functional programming language (for PACT). Also, they experimented with porting existing high level declarative languages to Stratosphere. One major research was unifying the Sopremo and PACT layer into a single model, that should support user defined functions as well as operators with known semantics. Another major area was defined as improving fault tolerance by introducing an adaptive algorithm, that picks the intermediate to materialize and is able to take different statistics into account. To the best of our knowledge, many of those goals have been implemented in later Stratosphere versions (support for functional programming language SCALA [10]) and afterwards Apache Flink [8].

## IV. Outlook

Aurora, as part of the Borealis project that added support for parallel and distributed execution, was commercialized as StreamBase Inc. in 2003 and was the first commercial real-time stream processing engine [10]. The company provides an enterprise data stream management system and was bought by TIBCO in 2013 [13], which continues to offer enterprise solutions for streaming applications. [11]

Members of the Stratosphere project continued with publications until 2016. With the transition to Apache Flink,

[4]https://hadoop.apache.org/
[5]https://spark.apache.org/
[6]https://www.microsoft.com/en-us/research/project/dryad/
[7]https://hadoop.apache.org/docs/r1.2.1/index.html#MapReduce
[8]https://hive.apache.org/
[9]https://giraph.apache.org/
[10]https://dms.sztaki.hu/sites/dms.sztaki.hu/files/file/2014/stratosphere_meetup.pdf. Last accessed on 06.07.2019. In this presentation, the author uses SCALA to present the system.
[11]https://www.tibco.com. Last accessed on 09.07.2019

the project now supports stream and batch processing. The team followed their research goals, which we described in Chapter III-H, and mainly focused on parallelization and optimization topics [11]. Parallel to the academic Stratosphere project, some of the authors founded "data artisans", the first commercial installment that offers an enterprise solution based on Apache Flink, and continues to contribute to Flink's open source development. Data artisans was renamed to Ververica [14] and sold to Alibaba in January 2019 [15]. These developments and the large industrial user base show that the Stratosphere project was hugely successful.

In the process of our work on stream and batch processing, we noticed an interesting dichotomy of processing data as a stream and as a batch. Aurora, a data stream management system, generates small batches of data  tuple trains - before processing them. The developers call the principle *train scheduling*. Further, the system tries to pipeline execution of each tuple train over as many operators as possible, which is called *superbox scheduling*. This means that the self-declared streaming platform relies on small batches of data for efficient processing.

Stratosphere, on the other hand, was developed as a batch processing system. Interestingly, Stratosphere's execution engine aims to process each data tuple as soon as it arrives, which can lead to some tuples being several operations ahead of other tuples from the same data set. Therefore Stratosphere decomposes a batch and, at least on each Task Manager, creates a pipelined stream out of it.

There is a third processing principle regarding big data, called micro-batching, which treats arriving data as small batches and then processes those small batches together. This technique is employed by popular Apache Spark. In addition, Apache Flink allows stream- as well as batch processing with the same engine.

Following those observations, it seems that for big data applications, a combination of treating data situationally as a stream- and a batch, is the key for a high performing system. Incoming batches may be partly formed into a stream and vice versa.

## V. Related Work

In 2013, authors from University of California published their research on *discretized streams* (D-Streams) [16]. They implemented their work as part of Apache Spark. As already mentioned, data is collected into so called *micro batches*, which means that several tuples in a pre-defined, sub-second-sized window get grouped together into an immutable data set. A stream of *micro batches* is called a D-Stream. Each *micro batch* is then distributed to a worker node within a cluster by a master node. The worker performs a series of operations on the dataset, thus creating a new immutable data set. In this way, each operation is stateless.

This system provides some interesting advantages to the continuous stream processing model of Aurora (and its later iterations, i.e. Borealis and StreamBase). Regarding recovery, the system is able to track the lineage of operations, that were performed on each *micro batch*. This is possible, because of a data structure called *Resilient Distributed Datasets* (RDD), that stores the transformations that created the data set. Because the intermittent results are stored in-memory and supersede the replication of data for fault tolerance, recovering times are sped up significantly. Additionally, also because replication is not necessary, the end-to-end processing times are increased drastically.

Beside having a faster and more resilient performance than systems with continuous operator systems, D-Streams have one downside. By intentionally delaying the processing of arriving tuples, the minimum end-to-end processing delay is raised. This makes micro batching not suitable for applications with sub-second response requirements, such as some stock trading applications. This issue is adjustable to some degree, because users are able to define the window for creating micro batches by themselves.

The D-Stream approach makes use of earlier research on stream processing, and subsequently focuses on the weaknesses of earlier systems, such as recovering mechanisms and parallel execution. The performance increases of using D-Streams instead of other systems are remarkable, which partly explains the ongoing popularity of Apache Spark [17].

## VI. Conclusion

With the amount of digital data drastically increasing, new use cases and business models connected to handling data have arisen. Two ways of analyzing data stand out: the processing of continuous data flows, so called streams, and the processing of large batches of data at once. Traditional RDBMSs were not designed for and therefore show poor performance in those tasks. In this report we presented two systems that address the issues with processing big data. Aurora is a data stream management system from 2003 which presents fundamentally different approaches to RDBMSs with regards to how data streams are handled. The main feature is the boxes-and-arrows system that is able to perform analysis of continuous streams, as well as views and ad-hoc queries, in real time. It has, like Stratosphere, fundamentally different assumptions from relational database management systems about the data that it processes. Stratosphere, on the other hand, focuses on batch processing. The system's main features include in-situ data processing and an extensible operator set with access to different abstraction layers, which is a generalization of MapReduce [4] by offering more operators and native support for iterative operations. It also features automated optimization and parallel, distributed execution. In our opinion, both systems were successful, because they made the right assumptions about the kind of data processing tasks that they wanted to address, and consistently followed these assumptions in their respective system design.

## References

[1] https://www.washingtonpost.com/news/the-switch/wp/2018/04/04/facebook-said-the-personal-data-of-most-its-2-billion-users-has-been-collected-and-shared-with-outsiders/?noredirect=on&utm_term=.5c21c59a9502, accessed: 2019-07-09.

[2] https://netzpolitik.org/2018/cambridge-analytica-was-wir-ueber-das-groesste-datenleck-in-der-geschichte-von-facebook-wissen/, accessed: 2019-07-09.

[3] A. Floratou, N. Teletia, D. J. DeWitt, J. M. Patel, and D. Zhang, "Can the elephants handle the nosql onslaught?" *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1712–1723, Aug. 2012. [Online]. Available: http://dx.doi.org/10.14778/2367502.2367511

[4] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[5] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 165–178. [Online]. Available: http://doi.acm.org/10.1145/1559845.1559865

[6] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, Aug 2003. [Online]. Available: https://doi.org/10.1007/s00778-003-0095-z

[7] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke, "The stratosphere platform for big data analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, Dec. 2014. [Online]. Available: https://doi.org/10.1007/s00778-014-0357-y

[8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[9] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM computing surveys (CSUR)*, vol. 15, no. 4, pp. 287–317, 1983.

[10] U. Çetintemel, D. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, M. Cherniack, J.-H. Hwang, S. Madden, A. Maskey, A. Rasin, E. Ryvkina, M. Stonebraker, N. Tatbul, Y. Xing, and S. Zdonik, *The Aurora and Borealis Stream Processing Engines*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 337–359. [Online]. Available: https://doi.org/10.1007/978-3-540-28608-0_17

[11] "List of publications originating from stratosphere project." https://stratosphere.eu/project/publications/.

[12] "Description of stratosphere project by participating institute." https://www.dima.tu-berlin.de/menue/research/completed_projects/stratosphere_information_management_above_the_clouds/, accessed: 2019-07-04.

[13] "Press statement on tibco buying streambase inc." https://www.tibco.com/press-releases/2013/tibco-software-acquires-streambase-systems, accessed: 2019-07-04.

[14] https://www.ververica.com/about, accessed: 2019-07-04.

[15] "Press statement on alibaba buying ververica." https://www.ververica.com/blog/data-artisans-alibaba-new-chapter-for-open-source-big-data, accessed: 2019-07-04.

[16] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 423–438. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522737

[17] "Apache spark overview of users." https://spark.apache.org/powered-by.html, accessed: 2019-07-09.