# How to Fit when No One Size Fits

Harold Lim
Duke University
harold@cs.duke.edu

Yuzhang Han
Duke University
yuzhangh@cs.duke.edu

Shivnath Babu
Duke University
shivnath@cs.duke.edu

## ABSTRACT

While "no one size fits all" is a sound philosophy for system designers to follow, it poses multiple challenges for application developers and system administrators. It can be hard for an application developer to pick one system when the needs of her application match the features of multiple "one size" systems. The choice becomes considerably harder when different components of an application fit the features of different "one size" systems. Considerable manual effort goes into creating and tuning such multi-system applications. An application's data and workload properties may change over time, often in unpredictable and bursty ways. Consequently, the "one size" system that is best for an application can change over time. Adapting to change can be hard when application development is coupled tightly with any individual "one size" system.

In this paper, we make the case for developing a new breed of Database Management Systems that we term *DBMS*$^+$. A DBMS$^+$ contains multiple "one size" systems internally. An application specifies its execution requirements on aspects like performance, availability, consistency, change, and cost to the DBMS$^+$ declaratively. For all requests (e.g., queries) made by the application, the DBMS$^+$ will select the execution plan that meets the application's requirements best. A unique aspect of the execution plan in a DBMS$^+$ is that the plan includes the selection of one or more "one size" systems. The plan is then deployed and managed automatically on the selected system(s). If application requirements change beyond what was planned for originally by the DBMS$^+$, then the application can be reoptimized and redeployed; usually with no additional effort required from the application developer.

The DBMS$^+$ approach has the potential to address the challenges that application developers and system administrators face from the vast and growing number of "one size" systems today. However, this approach poses many research challenges that we discuss in this paper. We are taking the DBMS$^+$ approach in a platform, called *Cyclops*, that we are building for continuous query execution. We will use Cyclops throughout the paper to give concrete illustrations of the benefits and challenges of the DBMS$^+$ approach.


$6^{th}$ *Biennial Conference on Innovative Data Systems Research (CIDR '13)* January 6-9, 2013, Asilomar, California, USA.

## 1. INTRODUCTION

The "no one size fits all" philosophy of system design has led to a variety of systems being developed in recent years. Examples include NoSQL systems, column-stores, MapReduce, data stream managers, complex event processors, in-memory databases, and others. While "no one size fits all" is a sound philosophy for system designers to follow, it poses challenges for the application developer or system administrator under three situations:

- *More choices could mean harder decisions:* All too often, the features of multiple "one size" systems may fit the primary needs of an application. The application developer can have a tough time deciding which system to use: should she use a SQL system or a NoSQL system? within NoSQL, should she use a key-value store or a column-family-oriented system or a document-oriented database? and so on. Benchmarking different systems is not easy, especially when the application has not been developed fully.
- *Jack of all trades or the masters of each?* Sometimes an application can have multiple components that have very different execution requirements. (Illustrative examples are given shortly.) In these cases, the application developer has to decide whether to use one system that best matches these different requirements or to use multiple "one size" systems that are individually best for the various components of the application. The system administrator may prefer the single-system option because she has less systems to manage. However, the multi-system option may give superior performance.
- *Change is inevitable:* The execution requirements of an application can change significantly over time, sometimes unpredictably. A fairly common scenario today is one where an application becomes very popular over a short time period, and its workload increases by many orders of magnitude. In such cases, different "one size" systems may be best for the application at different points of time. If application development is coupled tightly with an individual "one size" system, then dealing with change can be slow and hard.

Let us consider *behavioral targeting (BT)* as an illustrative application. BT enables customized Web pages or advertisements to be shown to each user based on her past and current interactions with one or more Web sites [15, 59]. One major component of BT involves learning statistical machine-learning models for possibly millions of users from the terabytes of logs containing impressions (when a user is shown an advertisement), clicks (when a user clicks on an advertisement or URL), and searches. Another major component of BT involves using these models in conjunction with recent user activity to generate customized content in real-time. The first component matches the features provided by systems like parallel OLAP databases [39, 47, 53], MapReduce [26, 31, 21], column-

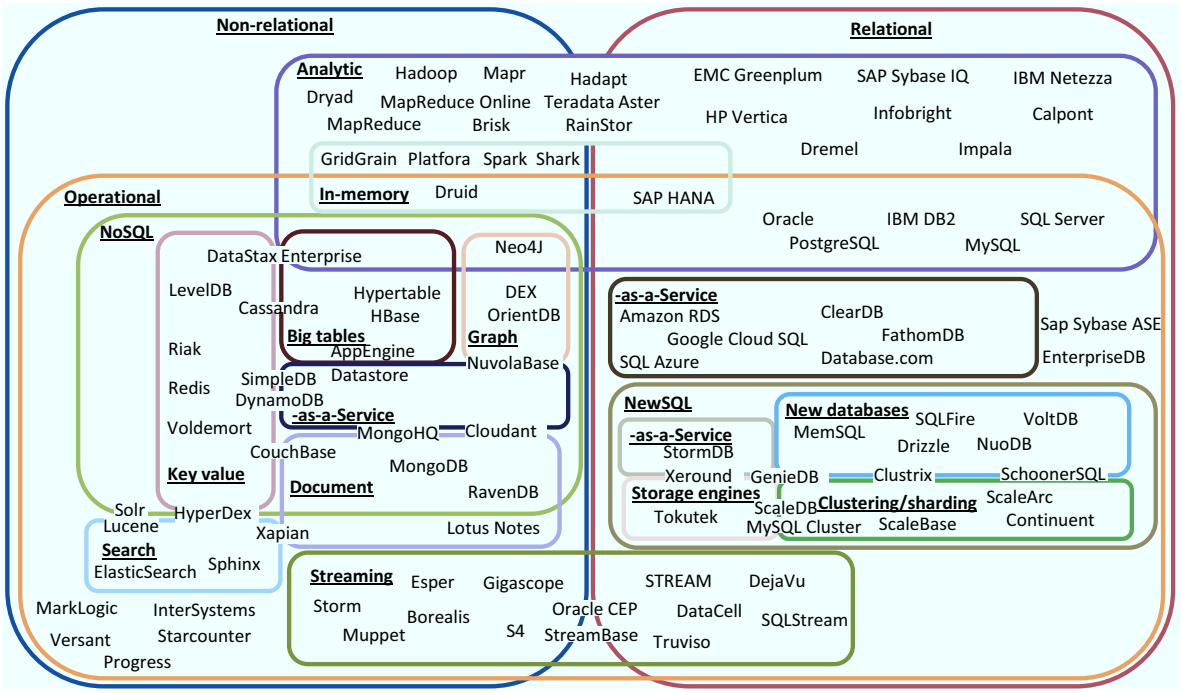


**Figure 1: A categorization of the variety of data systems available, many of which have been developed in recent years. This figure is an extension to a figure given in [7].**

family-oriented systems [40, 61], and in-memory analytics systems [52, 64]. The second component is a better match for low-latency systems like key-value stores [14, 27], sharded database clusters [41, 50, 62], and data stream processors [2, 23, 56, 9].

Another illustrative application creates the reports shown to the people who advertise on social networks, search engines, and microblogs [38]. One component of this application, by processing click and impression logs in real-time, shows advertisers how their advertising dollars are being spent. This component requires the generation of results in a predictable fashion and with low latency so that advertisers can adjust their advertising campaigns quickly when needed. Aggregate (approximate) results are acceptable here. At the same time, a second component is billing advertisers for user clicks. Here, advertisers want 100% accuracy, and care much less about low-latency results.

The above two examples come from a growing class of applications where different components of an application fit the features of different "one size" systems. In addition, the primary needs of any individual application component may also match the features of multiple "one size" systems. Considerable manual effort goes into creating, tuning, and maintaining these multi-system applications today. In addition, it becomes nontrivial to keep the applications running smoothly as data, workload, and system properties change over time, often in unpredictable and bursty ways.

One research approach to tackle this problem is to take a "one size" system that is designed to do $X$, and then study how the system can also be made to do $Y$ with or without significant modifications. Some recent examples of this approach include: (i) *MapReduce Online*, which adds pipelining and stream processing capabilities to the batch-oriented MapReduce [18]; (ii) *Online Aggregation* for MapReduce, which adds approximate answering capabilities to MapReduce [46]; and (iii) *HadoopDB*, which enhances MapReduce's performance for SQL query processing while retaining MapReduce's fault-tolerance and fine-grained adaptivity [3].

In this position paper, we make the case for a different approach that calls for developing a new breed of Database Management

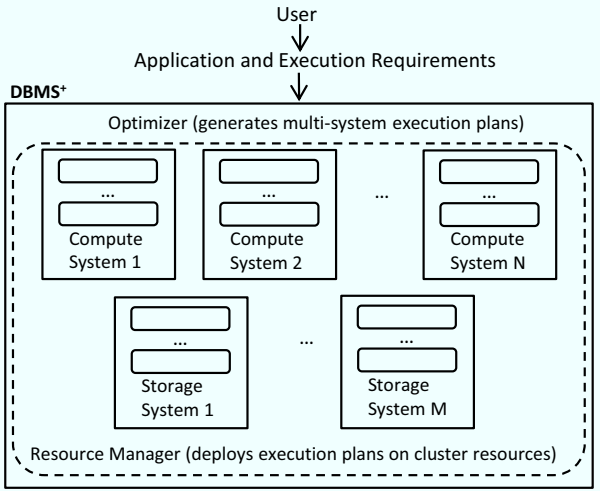


**Figure 2: An architectural overview of a DBMS$^+$.**

Systems that we term *DBMS*$^+$.[1] Figure 2 is an illustration of a DBMS$^+$. An application interacts with a DBMS$^+$ like it interacts with any conventional DBMS. For example, the application may issue SQL queries. However, there is one crucial difference. In a DBMS$^+$, the execution requirements for the application—e.g., tolerable latency bounds on query performance or desired recovery time under faults—are also specified declaratively along with the application.

A traditional DBMS usually has one execution engine and one storage engine. Typically, these two engines are coupled tightly with each other, and are not usable individually. In contrast, a DBMS$^+$ contains multiple "one size" compute and storage systems internally. Like the query optimizer in a DBMS, the *Optimizer* in the DBMS$^+$ (see Figure 2) has the responsibility of determining the best *execution plan* to serve requests made by the application.

[1]The name DBMS$^+$ represents "Data Management System for Multiple Systems." The S$^+$ notation is a use of regular expression syntax to denote one or more systems.

A novel aspect of the execution plan in a DBMS$^+$—which is another key difference between a DBMS and a DBMS$^+$—is that the plan includes the selection of one or more "one size" compute and storage systems that can best serve the application's requirements. Broadly speaking, a DBMS$^+$ commoditizes compute and storage systems and treats them in the same way that a DBMS will treat a physical operator like an index scan or a hash join.

The *Resource Manager* in the DBMS$^+$ is responsible for deploying the execution plan on cluster resources by running the "one size" systems selected by the plan. The execution of the plan is monitored and managed automatically. If application requirements were to change beyond what was planned for originally by the DBMS$^+$ Optimizer, then the application can be reoptimized and redeployed; usually with no additional effort required from the application developer.

Some crucial trends favor the DBMS$^+$ approach. First, system-independent, declarative languages like SQL continue to be preferred by application developers for applications to interact with data systems compared to low-level or system-specific approaches like MapReduce and programming languages. Second, data is getting stored more and more in system-independent serialization formats like *Avro* and *Protocol Buffers* [49]. System independence of query languages and data gives the DBMS$^+$ the freedom to choose the best system for execution and to migrate execution across systems as needed. The third trend is the growing popularity of database-as-a-service where application developers only care about getting their requirements met and not which database is used. Providers of database-as-a-service can use the DBMS$^+$ approach to reduce overall costs. A DBMS$^+$ has the advantage that each of the individual systems can be developed and improved independently as long as the system's external interfaces are preserved.

At the same time, the DBMS$^+$ approach poses a number of research challenges:

- Which "one size" compute and storage systems should be included in the DBMS$^+$ for a given application domain?
- How can the execution requirements of an application be specified declaratively?
- How can the DBMS$^+$ Optimizer automatically map a (sub)query $Q$ from an application to the compute system that is the best match for $Q$'s requirements?
- How can the DBMS$^+$ Optimizer automatically map a base, intermediate, or derived dataset $D$ to the storage system that is the best match for $D$'s access and maintenance requirements?
- How can the DBMS$^+$ Resource Manager coordinate the provisioning of compute and storage resources to the systems in an application-aware manner?
- When and how to perform automatic resource scaling, task migration from one system to another, and graceful load shedding under load spikes in an application-aware manner?

**Contributions and Roadmap:** In this paper, we will first make a case for the DBMS$^+$ approach by considering a concrete application domain, namely, continuous query processing [9]. At Duke University, we are building a continuous query processing platform called *Cyclops*. Cyclops instantiates the DBMS$^+$ approach by using multiple continuous query execution engines internally, namely, the *Esper* centralized streaming system, the *Storm* distributed streaming system, and the *Hadoop* system that is used popularly for batch analytics. We use examples from Cyclops throughout the paper to give concrete illustrations of the benefits and challenges of the DBMS$^+$ approach.

Section 2 introduces continuous query processing. Sections 3 and 4 drill down into continuous query processing in the context of the "no one size fits all" philosophy. In particular, we consider Esper, Storm, and Hadoop to show how each system outperforms the others in various regions of the continuous query processing spectrum. Section 5 gives a brief introduction to the architecture of Cyclops. Section 6 outlines a research agenda for applying the DBMS$^+$ approach to any application domain.

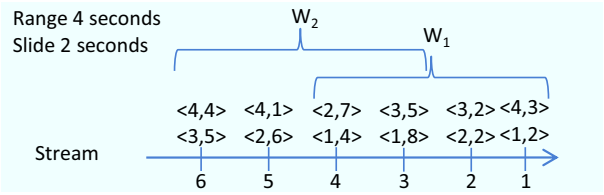


**Figure 3: An example showing two successive windows for a windowed aggregation query with Range = 4 seconds and Slide = 2 seconds. The two $\langle K,V \rangle$ tuples arriving every second are shown for time points 1 through 6.**

## 2. CONTINUOUS QUERIES

Processing a continuous query $Q$ will generate new query results whenever the data or time relevant to $Q$ changes. Continuous queries arise in a wide range of applications such as behavioral targeting, fraud detection, inventory management, network management, and environmental monitoring. Data stream managers and complex event processors are two popular examples of systems that process continuous queries.

For ease of presentation, we will focus on an important class of continuous queries called *windowed aggregation*. The following SQL-like syntax gives the template for a windowed aggregation query $Q$.

$Q$: Select $S.K$, *aggr*($S.V$)
From $S$ [Range $n$ seconds, Slide $d$ seconds]
(optional) Where *where_condition*
Group By $S.K$
(optional) Having *having_condition*

Here, $S$ is a stream of timestamped tuples. Each tuple, without loss of generality, contains two attributes $K$ and $V$. $K$ is the *grouping key* and $V$ is the *value attribute* on which the aggregation function *aggr* is performed per group. The optional Where and Having clauses specify filter conditions on the tuples before and after the grouping respectively.

The clause after $S$ in the query specifies a *sliding window* over the stream of tuples in $S$. This window is the main part that differentiates $Q$ from a conventional SQL query. The above template refers to a time-based sliding window with a Range of $n$ and Slide of $d$. At any point of time $t$, the Range parameter defines the interval of time over which tuples in the stream are part of the time-based window. Specifically, all tuples with timestamp in the interval $(t-n, t]$ are part of the window. Hence, the query result at time point $t$ is the result of the filtering, grouping, and aggregation over these tuples.

The Slide parameter specifies how the window advances over the stream over time. If the tuples in the interval $(t-n, t]$ constitute the current window, the next Slide will lead to a window with tuples in the interval $(t-n+d, t+d]$. We will illustrate windowed aggregation with the scenario shown in Figure 3 which we will also use as a running example throughout the paper.

Figure 3 shows a 6-second snippet of an example stream $S$. Two tuples with schema $\langle K,V \rangle$ arrive each second in the stream. Our example windowed aggregation query over $S$ specifies a Range of 4 seconds and a Slide of 2 seconds. $K$ is the grouping key. A sum aggregation is performed over the value attribute $V$ for every unique value of $K$ per window. There are no filtering conditions.

Figure 3 shows the tuples belonging to two successive windows $W_1$ and $W_2$. There is a 2-second overlap of tuples (4 tuples) between the two windows. For window $W_1$, there are three tuples with $K = 1$, and the sum of their $V$ fields is 14. There are two tuples each for $K = 2$ and $K = 3$, and the sum of their $V$ fields are 9 and 7 respectively. There is only a single tuple with $K = 4$, so the sum for this group is 3. The overall result of the query for $W_1$ will be <1,14>,<2,9>,<3,7>, and <4,3>. Performing the same computation for window $W_2$ yields the following result: <1,12>,<2,13>,<3,10>, and <4,5>.

The deceptively simple syntax for a windowed aggregation query can capture a wide range of application requirements including what has traditionally been considered batch analytics and what has been considered real-time (or streaming) analytics. To illustrate this point, we present three practical instances of windowed aggregation queries that arise in the context of a social networking Web site like Facebook.

- $CQ_1$: As part of behavioral targeting, the company wants to perform the aggregation of each user's activity for the past month, updated daily.
- $CQ_2$: As part of behavioral targeting, the company wants to track each user's unique clicks over the past 15 minutes, updated every minute.
- $CQ_3$: To keep track of the overall health of the Web site, the company wants to track the number of user logins from each region of the US over each five-minute interval.

$CQ_1$ has a Range of 1 month and a Slide of 1 day. Because of these large Range and Slide intervals, the amount of data per window and new data per Slide can run into many terabytes. Furthermore, a company like Facebook has hundreds of millions of active users. So, a query like $CQ_1$ that groups by user has to maintain a large number of unique entries per window.

Compared to $CQ_1$, $CQ_2$ has a much smaller Range of 15 minutes and Slide of 1 minute. Short Slide intervals indicate low-latency requirements. However, despite the shorter Range, the number of unique entries per window in $CQ_2$ can be of the same scale as in $CQ_1$. $CQ_3$ differs from $CQ_1$ and $CQ_2$ in that both the Range and Slide in $CQ_3$ have the same value of 5 minutes. Thus, there is no overlap between successive windows while processing $CQ_3$. In addition, the average size of each window in $CQ_3$ is also expected to be much smaller than in the other two queries.

# 3. EXECUTION PLANS FOR WINDOWED AGGREGATION

Continuous queries can be processed by different types of systems where each system has been designed to work well on a particular workload environment. Moreover, there are choices for continuous query execution that are independent of the systems used. In this section, we describe how the windowed aggregation query can be executed in different systems. We will continue to use our running example shown in Figure 3.

## 3.1 Centralized Streaming System

In the past decade, there have been a number of centralized systems designed for real-time stream and event processing. These systems have been designed with fast performance as the main priority. Thus, they store and process everything in the memory of a single node. One example of this type of system is Esper [23]. Specifically, Esper is a centralized complex event processing engine that is run as a single Java process. Esper provides a declarative language, APIs for implementing specialized operators, as well as other functions for custom handling of streams.

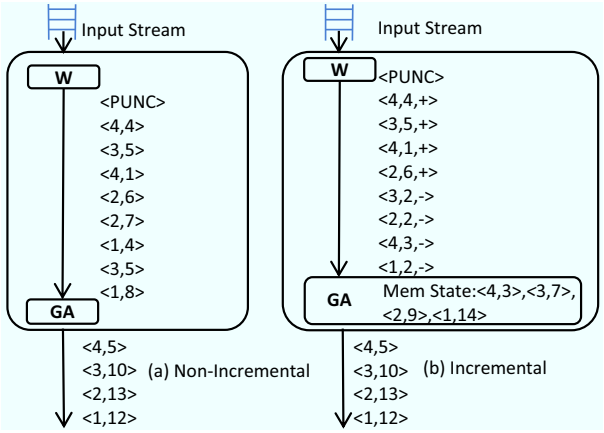


**Figure 4: An illustration of the operators in an Esper execution plan performing both non-incremental and incremental processing for window $W_2$ of the example stream.**

Figures 4(a) and (b) illustrate respectively how Esper executes the running example query *non-incrementally* and *incrementally*. In this figure and in describing the other systems in the remainder of this section, we focus on the processing of window $W_2$ in the example query. Esper has two operators running in pipelined execution: window operator (labeled W) and Groupby-Aggregation operator (labeled GA). Both of these operators are running continuously while the input stream feeds into the window operator. The window operator extracts tuples for each window and pushes them to the GA operator. Since streams are processed continuously, the GA operator needs a way to know the occurrence of a new window and to differentiate tuples between successive windows. Thus, the plan uses a punctuation tuple (labeled <PUNC> in the figure) that is created by the W operator and is sent to the GA operator.

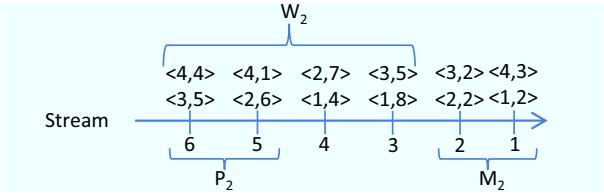


**Figure 5: An illustration of the insertions ($P_2$) and deletions ($M_2$) of tuples in window $W_2$ of the example from Figure 3.**

In non-incremental processing, all of the tuples within the window are extracted and sent to the GA operator. It also possible to process the query *incrementally* by only processing the differences between successive windows. Figure 5 shows the same example stream, but also shows the difference, in terms of insertions (or pluses, $P_2$) and deletions (or minuses, $M_2$) of tuples for window $W_2$ compared to $W_1$. Figure 4(b) shows how Esper process a query incrementally. Specifically, the W operator only sends the tuples in $M_2$ and $P_2$. In order to differentiate between the two types of tuples, the W operator also sets an additional meta information (marked by - and + in the figure). In the incremental processing plan, the GA operator maintains the previous aggregation results in memory (e.g., the results of $W_1$ are shown in the figure) as intermediate state.

Going back to the running example, the query results of $W_2$ can be computed by adding (subtracting) $V$ values for the tuples in $P_2$ ($M_2$) for each corresponding group in the query results of $W_1$ which was <1,14>,<2,9>,<3,7>, and <4,3>. For example, for $K = 1$, the $V$ field of <1,2> is subtracted from the $V$ field of the result of $W_1$ for $K = 1$: <1,14>, which results in <1,12>. Likewise, for $K = 2$, the $V$ field of <2,2> is subtracted from and the $V$ field of <2,6> is added to the previous window result <2,9>, which

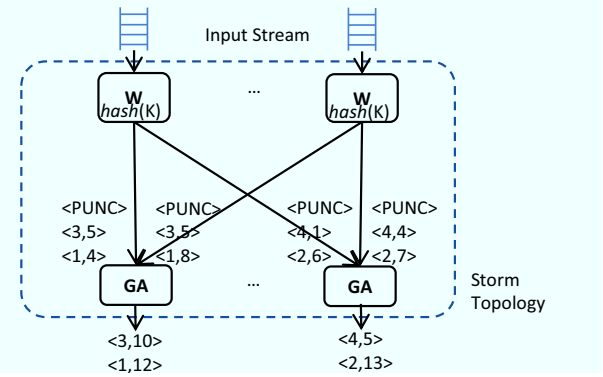


**Figure 6: Task-level illustration of a Storm execution plan performing non-incremental processing for window $W_2$ of the example stream.**

results in <2,13>. Performing incremental processing for the other groups yields the same results for window $W_2$ when processed non-incrementally: <1,12>,<2,13>,<3,10>, and <4,5>.

It should be noted that one way of processing a query is not always better than the other. Specifically, incremental processing of a query is beneficial when the differences between successive windows are small compared to the size of a window. However, in other cases where there is minimal to no overlap between windows, incremental ends up processing more data than non-incremental.

In both implementations, all of the states and computations are done in memory and within a single process. Furthermore, Esper is designed to utilize all the cores of a node for low-latency processing of the stream. However, since it runs on a single node and everything is kept in memory, Esper is not able to handle large amounts of intermediate state data. It also does not have native support for fault tolerance because none of the intermediate states or results are materialized to disk.

## 3.2 Distributed Streaming System

Distributed real-time stream processing systems are natural extensions to the previously mentioned type of systems. They share similar design goals in that they cater to queries with real-time requirements. However, this type of system also focuses on horizontal scalability, typically through a shared-nothing parallel architecture. Thus, distributed real-time stream processing systems can handle larger loads of data. Storm [56] is an example system here that has been gaining traction in the industry. Streams are processed by a user-defined plan (called *topology* in Storm's terminology) in a cluster of Storm nodes. Storm uses a push-based mechanism that pushes output tuples of one vertex to another in the topology. Common class abstractions are provided for defining how each vertex in the topology processes tuples of a stream as they propagate through the topology. Moreover, Storm provides APIs for specifying how vertices are parallelized (e.g., number of parallel tasks for each vertex) and for specifying how tuples are partitioned across tasks.

Storm is similar to Esper in that its implementation for the windowed aggregation query also has two types of operators: Window operator and Groupby-Aggregation operator. However, it also has native support for pipelined as well as partitioned parallelism where each operator runs multiple parallel tasks on different nodes. Figure 6 shows a task-level illustration of Storm for processing window $W_2$ of the example stream. The tasks in Storm are running continuously and the input stream feeds into the tasks of the W operator continuously. In this implementation, the input stream is distributed across multiple parallel W tasks. Partitioned parallelism is achieved by having a hash partition function in each of the W tasks. This function is applied to the output tuples of W to determine to which GA tasks to send each output tuple to.

Similar to the Esper implementation, the Storm implementation also uses punctuation tuples to notify the GA tasks of the occurrence of a new window. In contrast to the Esper implementation, the Storm implementation replicates and sends the punctuation tuple to each GA task. Moreover, since the underlying infrastructure of Storm does not provide any grouping and merging of tuples (i.e., tuples are pushed to another task as soon as they are produced), each GA task has to synchronize internally. Each GA task merges its input tuples by maintaining separate queues of input tuples from each W task, and uses punctuation tuples to determine which window each tuple belongs to. Also, each GA task waits until it has received punctuation tuples for each window from all W tasks before sending the results to the output stream.

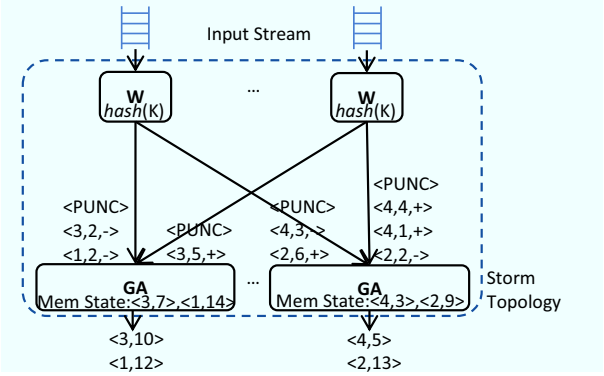


**Figure 7: Task-level illustration of a Storm execution plan performing incremental processing for window $W_2$ of the example stream.**

Figure 7 shows a task-level illustration of Storm for incremental processing for window $W_2$ of the example stream. The main difference in this implementation is that the W tasks only emit tuples of $M_2$ and $P_2$. Like the Esper implementation, each tuple has an additional +/- meta information to distinguish between insertion and deletion tuples. Similarly, the GA tasks also maintain the intermediate state, such as the previous window's results, in memory.

Similar to Esper, Storm is more suitable for a real-time continuous query workload. Since Storm can partition the stream to parallel tasks, it can handle streams with larger amounts of data compared to Esper. However, since Storm still keeps all intermediate state in memory, it will not be able to handle workloads that need to maintain large windows; but definitely larger than what Esper can handle. Since Storm is designed first for scalability rather than performance, Storm's performance on a single node is less efficient than that of Esper. Storm also has support for fault tolerance by providing an API for replaying input streams after faults.

## 3.3 Distributed Batch System

Perhaps surprisingly, another type of system that can run continuous queries is the type designed for batch processing of large amounts of data. MapReduce [21], Hadoop [26], and Dryad [31] are examples of such systems. In contrast to the two types of systems mentioned before, these systems are not meant for queries with low-latency requirements. However, these systems are designed to handle very large datasets by utilizing both memory and I/O resources as well as processing data in parallel using multiple tasks across a cluster. We chose Hadoop as the representative example of this type of system. Hadoop runs MapReduce jobs that are each specified by a map and a reduce function. During job execution, the input dataset is processed in parallel by a set of map

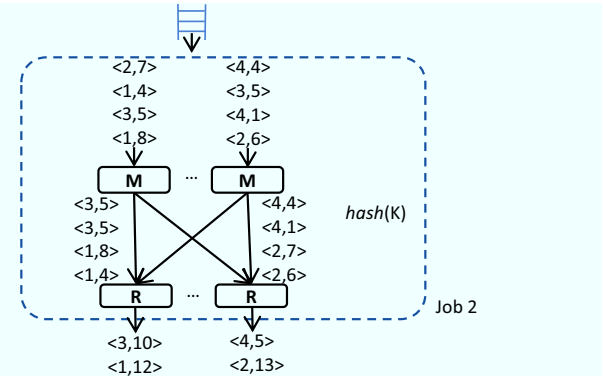


**Figure 8: Task-level illustration of a Hadoop execution plan performing non-incremental processing for window $W_2$ of the example stream.**

tasks. The output of map tasks are partitioned and processed in parallel by a set of reduce tasks. The outputs of tasks are always written to disk and are pulled by other tasks for further processing. Memory-intensive operations like sorting and grouping spill data to disk when dataset sizes exceed the available memory.

In contrast to the previous two systems, Hadoop's tasks are not running continuously. When a job is submitted to Hadoop, tasks are scheduled, launched, and terminated when they are done. In Hadoop, each window is processed by a separate MapReduce job. Figure 8 shows a task-level illustration of a Hadoop job processing window $W_2$ of the example stream. One main difference is that Hadoop does not directly process an input stream. Instead, a control program launches the jobs for each window. This program sets the input dataset for each job. In the figure, the input dataset comprises the tuples in $W_2$ of the stream.

Hadoop has native support for partitioned parallelism and materializes all outputs to disk. The input dataset is split and processed in parallel by map tasks. Similarly, output tuples of map tasks are partitioned, grouped, and merged by the underlying Hadoop infrastructure. Thus, the input to the reduce tasks are already grouped by the grouping field. In the Hadoop implementation that does non-incremental processing, the map tasks simply set the grouping key of each tuple as the tuple's map output key. The reduce tasks perform aggregations for each unique grouping key that they receive.

Figure 9 shows a task-level illustration of a Hadoop job performing incremental processing for window $W_2$ of the example stream. In contrast to Storm, intermediate state such as the previous results are not stored in memory because tasks are not running continuously. Thus, the inputs to the Hadoop job are of three types: $M_2$, $P_2$, and the results of the previous job. In this case, the map tasks also tag each input tuple based on which type it belongs to (+, -, and J1 in the figure). When the reduce tasks receive the tuples for each group, they apply the tuples with either +/- tags to the previous result for this group (i.e., tuples with J1 tag).

Hadoop is designed for processing big data; thus, it can handle very large windows of streams. Moreover, Hadoop utilizes both memory and I/O through spilling of tuples to disk during grouping and sorting of tuples, which allows it to handle queries with large intermediate state. However, since many tasks have to be scheduled, launched, and run on the cluster, there is overhead for starting up and cleaning up each MapReduce job (around 30 seconds). Thus, Hadoop is not ideal for close to real-time continuous query workloads. However, since output tuples are always materialized to disk, Hadoop can simply restart tasks when failures occur.

# 4. ANALYSIS AND EVALUATION

As mentioned previously, windowed aggregation queries have a wide spectrum of applications. The most suitable execution plan

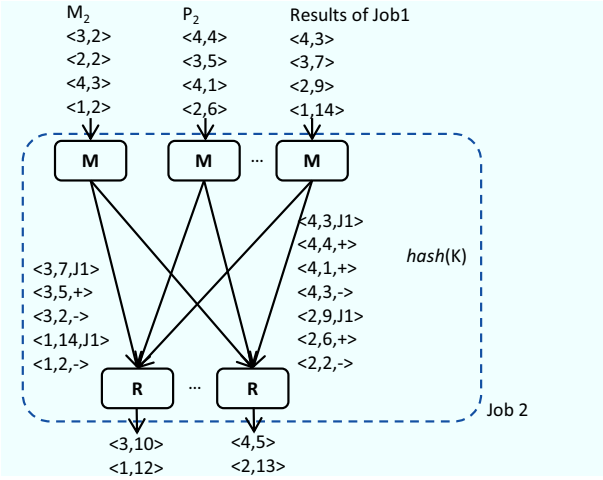


**Figure 9: Task-level illustration of a Hadoop execution plan performing incremental processing for window $W_2$ of the example stream.**

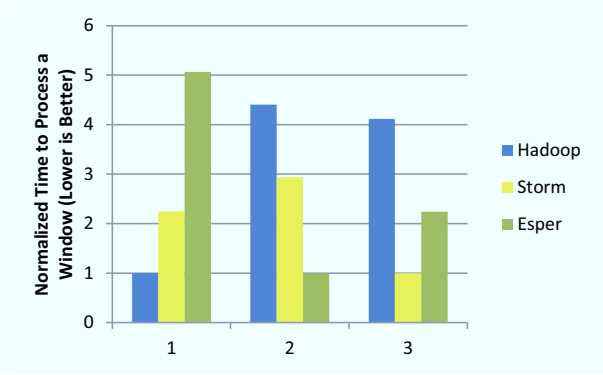


**Figure 10: The performance of Hadoop, Storm, and Esper on different workloads.**

to execute a window aggregation query varies depending on the characteristics of the query. Specifically, we identify two characteristics: i) window specification, which is defined by the Range and Slide parameters, as described previously, and ii) stream arrival rate, which describes the number of incoming tuples for each unit of time and affects the amount of data per window.

In this section, we evaluate the performance of Esper, Stream, and Hadoop for processing windowed aggregation queries. The goal of our evaluation is to motivate and show that no one system dominates the others. In the experiments, both Hadoop and Storm are run on a 11-node m1.large Amazon EC2 cluster. Esper is run on a single m1.large Amazon EC2 node. The stream tuples are pre-generated before each experiment, stored in the Hadoop Distributed Filesystem (HDFS), and replayed during the experiment.

Figure 10 shows the performance (normalized time to process a window) of Hadoop, Storm, and Esper for processing windowed aggregation queries. In each system, we show the performance of the technique (incremental Vs. non-incremental) that results in the best performance. The queries executed in this experiment are similar to our running example of performing summation on a window, but with different stream arrival rates and window specifications. Each tuple in the stream consists of a $K$ field and a $V$ field. The queries group on the $K$ field and sum the $V$ field.

The results show that "no one size fits" even with the same type of windowed aggregation query. Specifically, each individual system performs better and also worse than the other two systems (by at least 4x) depending on the characteristics of the query.

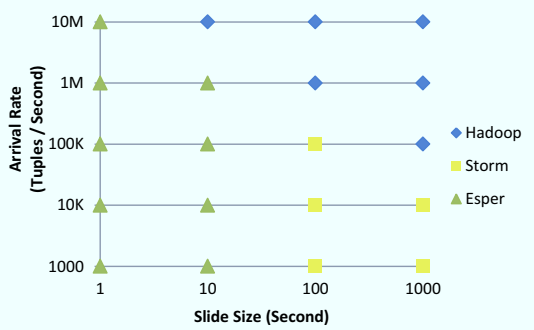


**Figure 11: Two-dimensional grid showing which of the three compute systems performs best on a given combination of arrival rate and Slide size. Range is set the same as the Slide.**

In the first query (labeled 1 in the figure), Hadoop outperforms both Storm and Esper. Analyzing the characteristics of this query explains the result. In this query, the window specifies a Range of 1 hour and a Slide of 10 minutes. Moreover, the stream has an arrival rate of 220 thousand tuples per second. In a Slide of 10 minutes, this arrival rate results in 132 million new tuples in each Slide and 792 million tuples in each window. Thus, it is not surprising that Hadoop performs better because of the size of the data, which Hadoop can efficiently process using compute and storage resources. On the other hand, Esper performs the worst because it is not able to partition and parallelize the processing of the stream.

In contrast, Esper outperforms Storm and Hadoop in the second query. This query has a real-time requirement, with a Range of 5 seconds and Slide of 1 second. The stream has an arrival rate of a million tuples per second, which is small enough for Esper to handle efficiently. On the other hand, Hadoop and Storm have additional overheads—such as job startup costs in Hadoop and network communication in both systems—due to the focus on scalability in their design. Note that running Storm on a single node does not eliminate these overheads. Specifically, when the second query is run on a single Storm node, the time to process each window is 6x worse than that for Esper.

The third query has similar real-time properties as the second query, but with a slightly larger Range of 30 seconds and Slide of 15 seconds. Thus, Hadoop still performs worst out of the three systems. However, in this query, Storm is able to perform better than Esper. The increase in Range and Slide results in larger size of data for each window, which Storm partitions and processes across multiple tasks in parallel.

To further show that "no one size fits" for processing continuous queries, we ran windowed aggregation queries with different combinations of characteristics. Specifically, we varied the arrival rate of tuples from 1000 to 10 million tuples per second and the Slide (and Range) parameter from 1 to 1000 seconds. Figure 11 shows the region of the space where each compute system performs better than the others.

# 5. CYCLOPS

So far we saw multiple systems for continuous query processing and how each system works well for a specific range of execution requirements. Similar results have been shown for other application domains, e.g., those served by NoSQL systems [19]. These observations lead naturally to the DBMS$^+$ approach. In this section, we introduce the Cyclops DBMS$^+$ system that we are building for continuous query processing. The next section will use Cyclops to give concrete illustrations of the benefits and challenges of the DBMS$^+$ approach.

Figure 12 shows a deployment of Cyclops to support continuous analytics in a multi-tiered Web service. Notice that the deployment has a cyclic nature of data flow among the tiers. Going back to the behavioral targeting application introduced in Section 1, the front-end tier has systems (e.g., Web, application, and cache servers) responsible for displaying content to users in real-time. In order to personalize the content displayed to users, the front-end tier relies on user models that are generated by the back-end cluster.

Building these models requires information about each user. Thus, user-activity logs are extracted from the front-end tier and streamed to an in-memory queuing system (e.g., *Kafka*) or a key-value store (e.g., *HBase*). Continuous queries are run on the back-end cluster that perform data analytics, such as windowed aggregation and machine learning, to build the models. The generated models may be stored in a low-latency key-value store.

Today, considerable manual effort goes into setting up a deployment like Figure 12 that can support different types of execution requirements on continuous queries. (Recall the example queries $CQ_1$–$CQ_3$ from Section 2.) For example, an application developer may choose to write queries with low-latency requirements to run on a distributed streaming system such as Storm. Another developer may choose to run continuous queries using MapReduce since she may be concerned about future increases in intermediate data sizes. Very quickly, the back-end cluster ends up running a number of technologies like stream processing, MapReduce, custom scripts and Java code, and distributed storage; leading to considerable frustration in both application development (e.g., debugging) and operations (e.g., tuning and resource provisioning).

Cyclops addresses this problem using a DBMS$^+$ approach that brings all continuous query processing under a single, distributed, but centrally-managed, platform. The architectural overview of Cyclops is shown in Figure 12. Notice how this architecture is an instantiation of the general DBMS$^+$ architecture shown in Figure 2 with a specific set of "one size" systems for continuous query processing.

Cyclops exposes a common language for applications to express continuous queries and their execution requirements. This language, which is independent of the underlying compute and storage systems, is an extended version of the SQL-like template shown for windowed aggregation in Section 2. Currently, Cyclops supports the declarative specification of application execution requirements at the level of performance. Specifically, the Slide parameter of the continuous query specifies the latency requirement to process a window. For example, low-latency requirements are specified by having a short Slide.

When a continuous query is submitted, Cyclops' Optimizer takes the specified requirements into account to select the best execution plan. The execution plan space is a combination of logical choices like incremental Vs. non-incremental processing as well as system choices: Esper (a centralized streaming system), Storm (a distributed streaming system), and MapReduce (a distributed batch processing system). The Hadoop Distributed File System (HDFS) is supported as a shared storage for intermediate states/data. HDFS shares the same cluster resources as the compute systems.

After the Optimizer picks the best execution plan, Cyclops uses *YARN* [63] to provision cluster resources for the chosen system(s). YARN provides the mechanisms for running heterogeneous systems on the same cluster resources. The Resource Manager in Cyclops determines the amount of resources to allocate for running each execution plan.

# 6. DBMS$^+$ RESEARCH AGENDA

Designing and building a DBMS$^+$ leads to a number of research challenges. In this section, we identify six research topics and also discuss some directions for solutions.

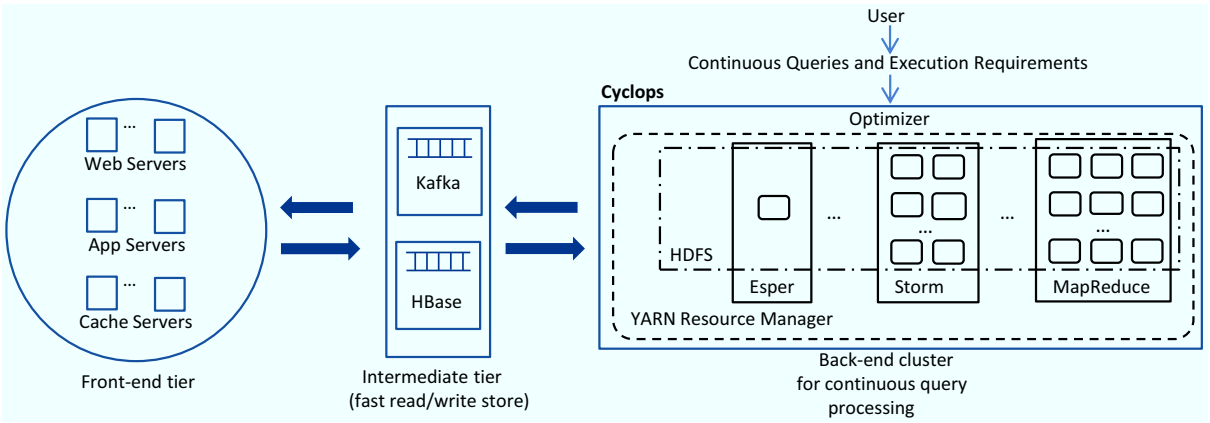


**Figure 12: The system architecture of Cyclops and its deployment for running continuous analytics on a typical multi-tiered scalable Web service.**

## 6.1 How to Integrate Systems into a DBMS$^+$?

One research challenge is to decide at which level of the system software stack should the DBMS$^+$ interact with and control. Figure 13 shows two approaches—*federated* and *imperial*—for how the DBMS$^+$ can interact with its internal "one size" systems.

In the federated approach (shown in Figure 13 (a)), the DBMS$^+$ integrates full-fledged query processing systems while exposing a common language to applications. These full-fledged systems tend to have their own languages and semantics. In this approach, each system is fairly autonomous and responsible for how queries are executed internally. The main advantage of this approach is that the DBMS$^+$ can be lean by reusing existing functionality of the underlying systems. *MaxStream* [12] is a research project that provides a thin federated layer on top of heterogeneous stream processing systems. However, one challenge in this approach is that semantic query execution differences between systems should be modeled and reconciled in order to achieve correct system integration. SECRET [13] attempts to solve this problem by providing a model for comparing the semantics of different stream processing systems.

With an imperial approach (shown in Figure 13(b)), the DBMS$^+$ directly uses the execution and/or storage engines of the underlying systems. The advantage of the imperial approach is that the DBMS$^+$ has full control of what gets executed and how. The DBMS$^+$ does not have to deal with semantic differences among the query languages supported by various systems. Cyclops uses the imperial approach for interacting with the underlying systems. MySQL's pluggable storage engine architecture is another use of the imperial approach [42]. MySQL hides the complexity of the underlying storage engines from applications by providing a common API. If application requirements change, then the underlying storage engine can be changed with no significant code changes.

Some trends in newer "one size" systems facilitate the imperial approach. Most of these systems make it easy for applications to specify exactly how queries should be executed. Furthermore, these systems are predominantly open-source, thus stripping away software layers to expose the raw execution and storage engines is possible.

## 6.2 Which Systems to Include?

Another challenge is to determine which systems to include in a DBMS$^+$. For example, windowed aggregation queries can be run by a wide range of systems ranging from pure streaming systems [23, 56] to batch processing systems [31, 26]. As our experimental results show, none of these systems dominates all other systems in all settings. However, this information alone does not help us decide which systems to include in Cyclops. Instead a methodology like the following is needed:

- Identify the spectrum of execution requirements for the application domain of interest
- Pick the set of systems whose features covers as much of the execution requirements as possible
- Eliminate redundant systems, i.e., systems that, in any setting, are outperformed by some other system

This methodology led us to pick Esper, Stream, and Hadoop as the systems to include in Cyclops.

## 6.3 Which Execution Plan to Pick?

The DBMS$^+$ Optimizer is responsible for picking the most suitable execution plan for a given query. The execution plan not only involves choosing techniques (e.g., incremental Vs. non-incremental processing) for running the query, but also choosing the most suitable system. Within each system, there may also be a number of parameters that affect the performance of a query. For example, Hadoop has many job-level configuration parameters such as the degree of parallelism and memory size for sorting. Thus, designing a multi-system query optimizer for a DBMS$^+$ is a nontrivial challenge. Moreover, the optimizer should be able to adapt to changes in application requirements over time.

The federated and imperial approaches lead to different optimization techniques. In a federated approach, each system usually has its own cost-based optimizer. Thus, the DBMS$^+$ can function mostly as a query rewriter that inputs the query or its subqueries to the per-system optimizers. The ASPEN project [36], which focuses on sensor network applications, uses this approach to optimize queries across different systems (e.g., sensor and stream engines). It is able to divide a query and decide which piece runs best on a particular system. This design enables a divide-and-conquer search strategy that allows each compute system to independently cost the execution plans. An advantage of this approach is that the DBMS$^+$ does not need a complex optimizer. However, the cost metrics used by one system's optimizer may not be the same as another system's optimizer. The DBMS$^+$ needs to ensure that the cost metrics obtained from different systems are compatible; if not, the DBMS$^+$ is responsible for reconciling the costs.

In contrast, an imperial approach does not depend on each system's optimizer, and hence, does not have to deal with incompatible costs. In the imperial approach, the DBMS$^+$ builds a cost model by itself for the execution plan space and uses the model to pick the best plan for a given query. One option for building models is to use white-box modeling which entails building analytical models

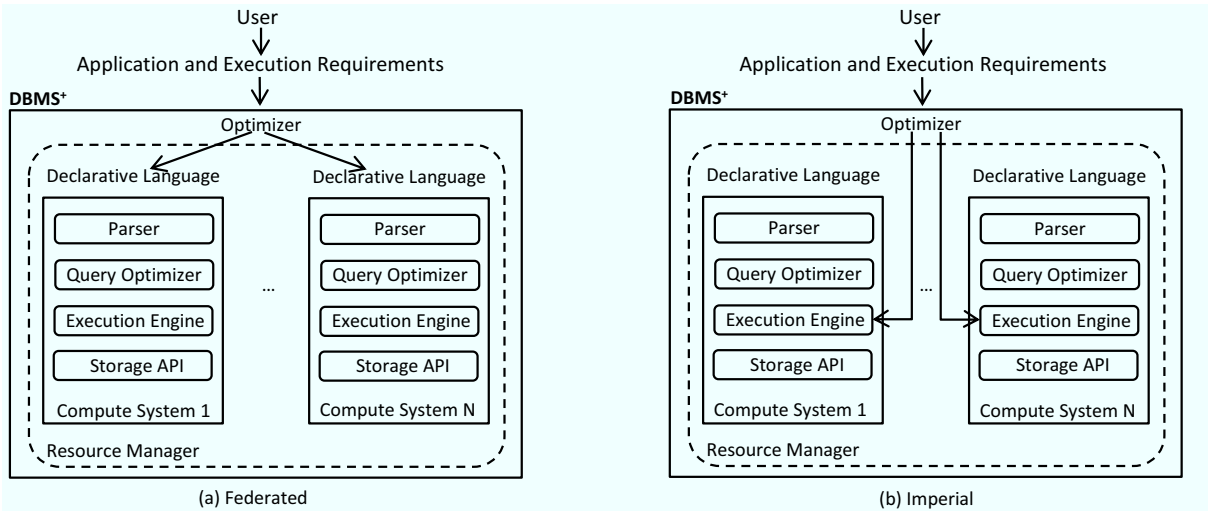


**Figure 13: An illustration of the (a) federated and (b) imperial approach for a DBMS$^+$ to interact with its internal "one size" systems.**

by understanding the internals of each system. However, the underlying systems can be complex and may need complex analytical models that are hard to develop and maintain.

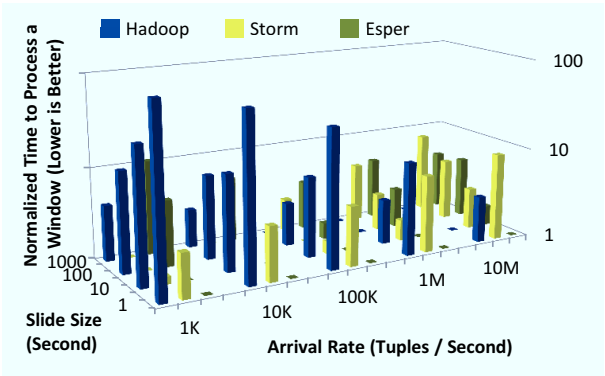


**Figure 14: A three-dimensional visualization showing the performance of Hadoop, Storm, and Esper with different combinations of stream arrival rate and Slide size.**

Another option is to use black-box modeling which involves three steps: (i) running selected benchmark queries on each system to collect training data; (ii) fitting models to this data in order to generate an initial cost model; and (iii) refining the model over time as more and more queries are executed [55]. All three steps pose interesting research challenges.

Cyclops uses black-box modeling to build cost models. For example, consider Figure 14. Windowed aggregation queries with different combinations of stream arrival rate (from 1000 to 10 million tuples per second) and Slide parameters (from 1 to 1000 seconds) are run on Esper, Storm, and Hadoop. The figure shows the normalized time to process a window in each case for each system. Cyclops applies regression techniques on the data collected from such runs in order to build the cost models.

## 6.4 How to Provision Resources?

With the popularity and ease-of-use of cloud-based platforms such as Amazon EC2 [22], there is a trend of running workloads in the cloud using pay-as-you-go cluster resources. Another emerging trend is to run multiple systems on the same cluster resources using platforms that can balance resource sharing and isolation [29, 63]. These platforms pose the challenge for the DBMS$^+$ to allocate cluster resources on demand to meet all application requirements. In addition to identifying the most suitable system to run queries, the ideal cluster or node-level configuration setting for running the system has to be found. Moreover, multiple queries can be running concurrently on different systems in the cluster.

The challenge of provisioning resources also involves dynamically adapting to changes in the query characteristics. Cloud-based services and resource managers (e.g., YARN resource manager) provide the mechanisms for elasticity. However, coming up with an effective policy to decide when to provision and the amount of resources to provision is a research challenge. Our previous work [35] attempts to address this challenge by designing a feedback controller to dynamically provision resources, but only focuses on provisioning for a single system. In contrast, the DBMS$^+$ must deal with provisioning multiple systems that are potentially sharing the same cluster resources; which, in conjunction with the DBMS$^+$ Optimizer, turns the challenge of resource provisioning into a multi-tenant, multi-query optimization problem.

## 6.5 How is the Data Stored?

Our case study focused on windowed aggregation queries where inputs and outputs are streams of data. In this case, the amount of data (state) maintained within each system is limited to the amount of data being processed in a window. However, other application domains may have different requirements. For example, consider ad-hoc queries that mine information from historical data. The DBMS$^+$ needs to manage how the data is stored and accessed by the systems for processing these queries.

One approach is to have a specialized and tightly-coupled storage system for each compute system. The advantage of this approach is that data can be stored and accessed efficiently by the compute system. However, interoperability among compute systems may be a challenge because of the custom storage formats used by each system. Furthermore, it is possible that migrating a query across systems requires copying data across systems and possibly converting from one data format to another format. Aside from the overhead of data transfer, maintaining multiple copies of data across systems is expensive.

Another approach is to have one or few global storage systems (e.g., HDFS, HBase) that can be accessed by all compute systems. With such a decoupled approach, the DBMS$^+$ does not have to migrate data across systems. However, accessing the data may not be as efficient for each system. Recent work has made accessing data on a global storage system more efficient, e.g., column stores on HDFS and serialization formats for storing structured data [49].

## 6.6 What are Application Requirements?

Cyclops currently focuses on performance as the only application execution requirement. Note that the results in Figures 10, 11, and 14 are all in terms of latency, namely, the time to process a window. Applications can have other execution requirements. We identify five types of requirements: performance, availability, consistency, cost, and dealing with changes.

A research challenge is to develop declarative abstractions and semantics for applications to specify their execution requirements. There has been progress in this direction in recent years. For example, the availability requirements for an application can be specified in terms of two properties: (i) *recovery time objective (RTO)*, which specifies the amount of time that the application can be down after a failure, and (ii) *recovery point objective (RPO)*, which specifies the data loss that can be tolerated after a failure. A related challenge is to provide continuous semantics for requirements that are seemingly boolean. For example, rather than simply specifying whether an application requires or does not require strict consistency, language abstractions can be developed that define consistency in terms of bounds on staleness of results.

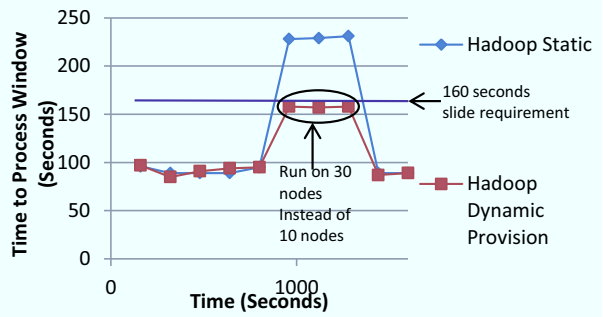


**Figure 15: The potential for adaptivity to bursts in stream arrival rate in Hadoop by temporarily provisioning more resources.**

Providing abstractions for applications to specify how they want to deal with changes can also be important. The DBMS$^+$ Optimizer can take such requirements into account and select execution plans that are more adaptive to changes. For example, Figure 15 shows the potential for adaptivity to bursts in stream arrival rate in Hadoop. Recall that the execution plan for a windowed aggregation query in Hadoop processes each window as a separate MapReduce job; which gives considerable scope for adaptivity. In Figure 15, the stream arrival rate starts out with 62.5K tuples per second and then jumps to 3000K tuples per second for around 10 minutes before going back to the original arrival rate. The query has Range and Slide of 160 seconds each. Without adaptivity, the time to process a window increases to 230 seconds from less than 100 seconds. Hadoop is able to adapt to the increase in stream arrival rate by temporarily provisioning more nodes to execute the query.

In contrast, Esper and Storm have continuously running tasks to process the query, making change a nontrivial exercise. The challenges that arise include coming up with query performance instrumentation mechanisms and policies for migrating from one plan to another, while ensuring little to no disruption or violation of the application requirements. A punctuation-based framework for stream processing systems was introduced in [54] that allows on-the-fly modifications of continuous queries.

It is also possible that the specified application requirements cannot be satisfied (e.g., not enough resources). A research challenge is to provide applications with results that they can still reason about, while degrading gracefully. Prior work introduces the concept of shedding load by dropping random tuples in the stream [8]. One issue with this approach is that it is difficult for applications to reason about the accuracy of the results. Thus, we take a different approach in Cyclops where load is shed by increasing the Slide parameter in the query. Intuitively, this approach preserves the accuracy of the result, but updates the result at a slower rate compared to when resources are plentiful. Specifically, Cyclops' Optimizer increases the Slide until a suitable execution plan can be found that satisfies the relaxed Slide requirement.

# 7. RELATED WORK

## 7.1 "One Size" Systems

**"One Size" Systems for Continuous Queries:** Aside from our choice of using Esper in Cyclops, there are a number of centralized stream processing systems available [2, 4, 20, 9]. The STREAM system [9] provides a continuous query language (CQL) that enables users to reason about streams using relational algebra semantics. Aurora [2] is another centralized stream processing system that provides users with an interface for specifying their applications through "boxes" and "arrows". It has mechanisms for load shedding to improve the quality of service (QoS). Both Aurora and STREAM perform dynamic query optimization (e.g., combining and reordering operators) by collecting run-time statistics.

In addition to Storm, there are a number of distributed real-time stream processing systems [1, 6, 5, 18, 33, 17, 45, 43, 51, 57, 25, 16, 60, 10]. Similar to Storm, S4 [43] is a distributed stream computing platform from Yahoo! that provides APIs for implementing processing elements. Borealis [1] is a follow-up project to Aurora and provides parallel processing of streams. It has an optimizer that uses local and neighbors' load information to balance the workload across a cluster by moving operators. StreamCloud [25] is a system that runs on top of Borealis. It transforms queries into subqueries that minimize the distribution overhead of parallel processing. TelegraphCQ [16] relies on adaptive routing to optimize query plans. It decides how to route data to different operators. It is also able to dynamically decide the optimal partitioning of streams for parallel processing.

There are also a number commercial systems that support distributed stream processing [5, 6, 45, 51, 57], each with its own specialization and differences. For example, Oracle CEP [45] is a Java container that works well with an event-driven architecture and provides both a native programming model and a declarative language for building an event processing network. System S [6] provides partitioned parallelism strategies that split streams into subgroups of split, aggregation, and join operators. The Sybase Event Stream Processor [51] provides users with a declarative language to specify their applications. Moreover, it is able to collect real-time statistics and monitor the performance of the cluster.

In addition to systems developed for real-time requirements, there are a number of other systems developed for batch processing of large amounts of data that can be used for processing continuous queries [28, 31, 11, 26, 30, 21, 37, 44, 64]. Comet [28] modifies Dryad [31] to have better support for batched stream processing by spliting a query into subqueries and reusing results of previous subqueries. Spark [64] is a cluster computing system for data analytics that provides abstractions and primitives for supporting in-memory computing. Recently, they have introduced a new primitive called d-streams that performs batch computations on small intervals of streams which avoids the overhead of per-tuple processing that is common in stream processing systems [65]. Nephele [37] extends parallel data processing systems, such as MapReduce [21], to handle streaming workloads with latency requirements. It automati-

cally adjusts the output buffer size and dynamically chains tasks based on run-time measurements.

**Other "One Size" Systems:** Note that Cyclops instantiates the DBMS$^+$ approach for management and execution of continuous queries. There are a large number of "one size" systems developed in recent years for other types of queries. Figure 1 shows a categorization of "one size" systems. For example, there are database systems that are either more suited for OLAP or OLTP workloads. Column-store databases, such as Vertica [61], can potentially perform better than row-store databases on OLAP workloads because of the large amounts of read operations. While traditional databases enforce strong consistency, there are new database systems, such as Cassandra [14] and Voldemort [48], that improve latency by having an eventual consistency data model.

## 7.2 Integration of Multiple Systems

The goal of this paper is to highlight the challenges of integrating different "one size" compute and storage systems for different types of applications. There is a lot of work related to systems integration, but this body of work focuses on some subset of the research challenges that we have identified. Tatbul [58] enumerates the challenges in integrating stream processing systems, such as the lack of a common semantic model across different systems, optimization challenges, and transactional issues. Recently, there was an attempt to standardize the language and semantics of different stream processing systems [32].

MaxStream [12] is a middleware that integrates heterogeneous stream processing systems, but lacks the optimizer for selecting the most suitable system. Similarly, MySQL integrates a number of storage engines through its pluggable storage engine architecture [42]. MySQL provides a common storage API that allows diverse storage engines to be used with MySQL. Moreover, MySQL encapsulates the implementation details of the storage engines from applications and queries. This feature allows applications and queries to be mapped to different engines based on their requirements. However, like MaxStream, MySQL lacks an optimizer for automatically selecting the most suitable storage engine based on the execution requirements of applications.

ASPEN [36] is another project that integrates multiple stream processing systems, but in the context of sensor networks. ASPEN has a federated optimizer for optimizing queries across systems, but lacks support for orchestration and management of systems.

Unlike the DBMS$^+$ approach to systems integration, a number of projects take a "one size" system that is designed to do $X$, and then study how the system can also be made to do $Y$ with or without significant modifications. For example, DataCell [34] is a stream processing engine that is built using a database kernel to take advantage of the existing algorithms and techniques in databases. Truviso [60] is a system that provides integrated stream and relational query processing.

## 7.3 Mechanisms for DBMS$^+$

Public and private cloud infrastructure providers, such as Amazon EC2 [22] and Eucalyptus [24], have provided a virtual machine abstraction for easily launching and running diverse "one size" systems. In addition, some recent projects provide a higher-level resource abstraction for running multiple distributed systems on shared cluster resources. Examples include the next generation Hadoop, which is also called YARN [63], and Mesos [29]. In contrast to the current version of Hadoop that supports MapReduce jobs only, YARN can run heterogeneous execution engines and applications concurrently on the same cluster resources. Mesos [29] is a cluster manager that handles fine-grained resource allocation and sharing across different systems such as MapReduce, high-performance computing (HPC) systems based on message passing interfaces (MPI), and in-memory analytics systems such as Spark.

These projects reinforce the "no one size fits all" philosophy and further motivate the need for a DBMS$^+$ that can automatically manage and launch components of an application on the most suitable system. While these projects provide the mechanisms for supporting our proposed DBMS$^+$, none of them focus on the policy questions, such as selecting the best systems to run a given query on, and determining the amount of resources to allocate for each system.

# 8. SUMMARY

In this paper, we described the challenges that application developers and system administrators face due to the "no one size fits all" philosophy of system design. Specifically, there are a number of "one size" systems available, and choosing the most suitable subset of systems requires considerable effort. Moreover, current applications are coupled tightly with individual systems, which makes dealing with changes to application execution requirements or workload characteristics nontrivial. Thus, we made the case for a new approach, called DBMS$^+$, for managing applications and their execution requirements on a number of "one size" systems.

While conventional Database Management Systems (DBMS) support applications with a single execution engine and a storage engine, a DBMS$^+$ integrates and manages multiple "one size" compute and storage systems. In this approach, users submit their application and execution requirements to the DBMS$^+$. The optimizer in the DBMS$^+$ is responsible for determining the most suitable execution plan, which includes choosing the most suitable compute and storage systems for a given application and its requirements. The DBMS$^+$ also has a resource manager that is responsible for managing and orchestrating the systems for running the applications.

However, building a DBMS$^+$ leads to a number of research challenges. These challenges include how the DBMS$^+$ interacts with its internal systems, how to select the systems to integrate, how to select the most suitable execution plan, how to provision resources, how the data is stored, and what application execution requirements to support. As a concrete instantiation of the DBMS$^+$ approach, we introduced the Cyclops platform for continuous query processing that we are currently building at Duke University. Cyclops manages and integrates three systems: Esper, Storm, and Hadoop, which can be categorized respectively as a centralized streaming system, a distributed streaming system, and a distributed batch processing system.

# 9. REFERENCES


[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB J.*, 12(2):120–139, 2003.

[3] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. In *VLDB*, 2009.

[4] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *PVLDB*, 5(10), 2012.

[5] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Kirilov, A. Ananthanarayan, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich,

B. Chandramouli, J. Goldstein, S. Bhat, Y. Li, V. D. Nicola, X. Wang, D. Maier, I. Santos, O. Nano, and S. Grell. Microsoft CEP Server and Online Behavioral Targeting. *PVLDB*, 2(2), 2009.

[6] H. Andrade, B. Gedik, K.-L. Wu, and P. S. Yu. Scale-Up Strategies for Processing High-Rate Data Streams in System S. In *ICDE*, 2009.

[7] M. Aslett. Updated database landscape graphic. http://blogs.the451group.com/information_management/2012/11/02/updated-database-landscape-graphic/.

[8] B. Babcock, M. Datar, and R. Motwani. Load Shedding for Aggregation queries over Data Streams. In *ICDE*, 2004.

[9] S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3), Sept. 2001.

[10] N. Backman, R. Fonseca, and U. Çetintemel. Managing Parallelism for Stream Processing in the Cloud. In *HotCDP*, 2012.

[11] D. Borthakur, J. Gray, J. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, et al. Apache Hadoop goes Realtime at Facebook. In *SIGMOD*, 2011.

[12] I. Botan, Y. Cho, R. Derakhshan, N. Dindar, L. Haas, K. Kim, C. Lee, G. Mundada, M.-C. Shan, N. Tatbul, Y. Yan, B. Yun, , and J. Zhang. Design and Implementation of the MaxStream Federated Stream Processing Architecture. Technical report, ETH Zurich, 2009. ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/6xx/632.pdf.

[13] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul. SECRET: a Model for Analysis of the Execution Semantics of Stream Processing Systems. *PVLDB*, 3(1-2), 2010.

[14] Apache Cassandra. http://cassandra.apache.org/.

[15] B. Chandramouli, J. Goldstein, and S. Duan. Temporal Analytics on Big Data for Web Advertising. In *ICDE*, 2012.

[16] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[17] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, 2000.

[18] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *NSDI*, 2010.

[19] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SOCC*, 2010.

[20] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *SIGMOD*, 2003.

[21] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[22] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.

[23] Esper. http://esper.codehaus.org/.

[24] Eucalyptus. http://www.eucalyptus.com.

[25] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, and P. Valduriez. StreamCloud: A Large Scale Data Streaming System. In *ICDCS*, 2010.

[26] Apache Hadoop. http://hadoop.apache.org/.

[27] Apache HBase. http://hbase.apache.org/.

[28] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: Batched Stream Processing for Data Intensive Distributed Computing. In *SOCC*, 2010.

[29] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[30] Apache Hive. http://hive.apache.org/.

[31] M. Isard, M. Budiu, and Y. Yu. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, 2007.

[32] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik. Towards a streaming SQL standard. *PVLDB*, 1(2), Aug. 2008.

[33] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan. Muppet: MapReduce-style Processing of Fast Data. *PVLDB*, 5(12), 2012.

[34] E. Liarou, R. Goncalves, and S. Idreos. Exploiting the Power of Relational Databases for Efficient Stream Processing. In *EDBT*, 2009.

[35] H. C. Lim, S. Babu, and J. S. Chase. Automated Control for Elastic Storage. In *ICAC*, 2010.

[36] M. Liu, S. R. Mihaylov, Z. Bao, M. Jacob, Z. G. Ives, B. T. Loo, and S. Guha. SmartCIS: Integrating Digital and Physical Environments. In *SIGMOD*, 2009.

[37] B. Lohrmann, D. Warneke, and O. Kao. Massively-Parallel Stream Processing under QoS Constraints with Nephele. In *HPDC*, 2012.

[38] N. Marz. How to beat the CAP theorem. http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html.

[39] Microsoft SQL Server Analysis Services. http://www.microsoft.com/sqlserver/en/us/solutions-technologies/business-intelligence/analysis.aspx.

[40] MonetDB. http://www.monetdb.org/Home.

[41] mongoDB. http://www.mongodb.org/.

[42] Overview of MySQL Storage Engine Architecture. http://dev.mysql.com/doc/refman/5.1/en/pluggable-storage-overview.html.

[43] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *ICDMW*, 2010.

[44] Oozie - Yahoo!'s Workflow Engine for Hadoop. http://yahoo.github.com/oozie/.

[45] Oracle CEP. http://www.oracle.com/technetwork/middleware/complex-event-processing/.

[46] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online Aggregation for Large MapReduce Jobs. *PVLDB*, 4(11), 2011.

[47] Pentaho Mondrian. http://mondrian.pentaho.com/.

[48] Project Voldemort. http://www.project-voldemort.com/voldemort/.

[49] Protocol Buffers. https://developers.google.com/protocol-buffers/.

[50] Redis. http://redis.io/.

[51] SAP Sybase Event Stream Processor. http://www.sybase.com/products/financialservicessolutions/complex-event-processing/.

[52] SAS In-Memory Analytics. http://www.sas.com/high-performance-analytics/how-does-it-work/in-memory.html.

[53] SAS OLAP Server. http://www.sas.com/technologies/dw/storage/mddb/index.html.

[54] K. Sheykh-Esmaili, T. Sanamrad, P. M. Fischer, and N. Tatbul. Changing Flights in Mid-air: A Model for Safely Modifying Continuous Queries. In *SIGMOD*, 2011.

[55] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, 2001.

[56] Storm. http://storm-project.net/.

[57] StreamBase. http://www.streambase.com.

[58] N. Tatbul. Streaming Data Integration: Challenges and Opportunities. In *NTII*, 2010.

[59] TellApart. http://tellapart.com/.

[60] Truviso. http://www.truviso.com.

[61] Vertica. http://www.vertica.com.

[62] Windows Azure SQL Database. http://msdn.microsoft.com/en-us/library/windowsazure/ee336279.aspx.

[63] Apache Hadoop NextGen MapReduce (YARN). http://hadoop.apache.org/docs/r0.23.0/hadoop-yarn/hadoop-yarn-site/YARN.html.

[64] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *HotCloud*, 2010.

[65] 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. In *HotCloud*, 2012.