ABSTRACT
Over the last few years, the increasing demand on processing streaming data with high throughput and low latency has led to the development of specialized stream processing engines (SPE). Although existing SPEs show high performance in evaluating stateless operations and stateful operations with small windows, their performance degrades significantly when calculating exact answers for complex aggregate queries with huge windows. Examples include correlated aggregations, quantile and ordering statistic computation. Meanwhile, modern database systems have demonstrated the ability of processing complex analytical tasks efficiently over very large datasets, using technologies such as vertical storage, vectorized query execution, etc. This suggests the feasibility of leveraging database systems to assist SPEs to process complex aggregate queries to reduce their evaluation latency.

The goal of this thesis is to investigate the potential of combining database systems with SPEs in the context of stream processing so as to improve the overall query evaluation performance. To this end, the following two major topics will be addressed in this thesis: (1) dynamic migration of complex aggregate operations between the SPE and the database in response to varying system load and (2) efficient evaluation of continuous queries over streaming data that is migrated to the database.

1. INTRODUCTION
The last decade has witnessed the pervasion of stream processing in a broad range of modern applications including sensor networks [24], stock market analysis [11], network monitoring [13] and so forth. These applications share a common characteristic, that high-velocity streaming data, which is continuously pushed to the applications, needs to be processed with sub-second latency.

Given the fact that queries in these applications are rather static (continuous queries [33]) compared to the fast moving data, it is naturally considered [5, 29] that the “store first, analyze later” model adopted in traditional database engines is inadequate to serve streaming applications due to the high latency penalty introduced by the storage operations. Instead, a different approach has to be used, in which data can be processed on the fly. Following this direction, a significant amount of effort has been spent on designing and building dedicated stream processing engines (SPE) in both academia [2, 9, 3] and industry [31, 32].

Data streams are potentially unbounded in size, making it impractical to evaluate continuous queries over the entire history of the data streams. One common technique for tackling this problem is to impose sliding windows on the streams which emphasize recent data over old data [5]. Sliding windows could be time-based or tuple-based. Both types of windows can be commonly characterized by the number of tuples contained in the window, which we define as the window-size. For tuple-based windows, the window-size can be directly derived from the window specification, while for time-based windows, the window-size is a derivative of the input data rate and the length of the time interval.

The state-of-the-art SPEs are able to evaluate operations with small windows efficiently even with naive operator implementations. Naive implementation means that operation results are computed by scanning the content of the window as many times as necessary. For operations with huge windows (e.g., windows containing several million tuples), low-latency responses can still be guaranteed by using intelligently implemented stream algorithms. The key idea behind these algorithms is either to calculate query results in a single-pass and incremental way [15, 12, 16], using online maintained synopsis (e.g., histograms [20], wavelets [17]), or to utilize a divide-and-conquer approach [22].

However, there exist a significant amount of applications requiring complex aggregate queries [12], for which these advanced techniques are either not applicable or only able to provide approximated answers. Examples of complex aggregations include quantile and order statistic computation, correlated aggregation, etc. The key characteristic of complex aggregate queries is that, query answers cannot be easily computed in an incremental way and usually a re-scanning over the data window is required. For correlated
aggregates, it even needs re-scanning the data window multiple times to compute the exact answers. However, approximate answers are not acceptable for applications in which the accuracy of query answers is of utmost importance, even compared to the latency of the answers. One example is decision support applications, where inaccurate query answers may lead to wrong business decisions and cause financial loss. To guarantee the accurate results of the complex aggregate queries, usually the only way is to return to the naive or near-naive implementations, which typically have high space and computation cost for huge windows. The consequence is that the evaluation latency degrades significantly from the microseconds to the seconds level.

While existing SPEs show limitations in computing exact answers for complex aggregates with huge windows, recent studies [6, 27] in the database community have shown that modern database systems are able to process very large datasets with second or even sub-second level latency, by leveraging vertical storage architecture, vectorized query execution, in-memory technology and so on. We took an example of correlated aggregation and compared its per-result evaluation time with increasing window size in a state-of-the-art commercial column-oriented in-memory database system and an SPE. The example is in the context of stock market analysis. It calculates the number of companies (comp) in each business area (bs_area) (e.g., retail, banking, insurance, etc.) whose market capitalization (mk_cap) is greater than 50% of the maximal market capitalization in this area in the past $k$ time units. This scenario can be expressed with relational algebra as follows:

$$\text{Comp}, \text{bs}_{\text{area}}, \text{mk}_{\text{cap}}, \text{timestamp}: R(c, b, m, t)$$

Data within $(t-k, t]$: $S(c, b, m) = \pi_{c,b,m} \sigma_{t-t+k \leq t} (R)$

Max. $\text{mk}_{\text{cap}}$ in each bs_area: $M(b, mb) = \pi_{b,mb} G_{\text{Max}}(mb)(S)$

Join between $S$ and $M$: $J(c, b, m, mb) = S \bowtie_{b=b} M$ \quad (S)

Final result: $C(b, c, mb) = \pi_{b,c,mb} G_{\text{Count}}(c) (\sigma_{mb > 0.5mb}(J))$

$t$ represents the current time.

The results are shown in Figure 1. We can observe that when the evaluation time in the SPE jumps to more than 3.5 seconds with a 5-million-tuple sized window, the column-oriented database is able to finish the same computation within 600 milliseconds.

Although the data transfer from the SPE to the database system would introduce additional latency, the SybaseESP engine [32] has shown that it is possible to push streaming data to database systems with high efficiency (several hundred thousand tuples per second). Therefore, even taking the overhead of data transfer into account, we can still find a window size beyond which the per-result evaluation time of the operation in the database is smaller than that in the SPE.

This result suggests the possibility of leveraging database systems to assist SPEs for processing complex aggregate queries to reduce the response time when the window size is huge and the accuracy of answers is a must. Considering that the performance of SPEs is still superior to database systems when processing stateless operations and stateful operations with small windows, in this PhD thesis, we propose a prototype system which combines the usage of a database system and a SPE for stream processing, and migrates complex aggregate operations with huge windows from the SPE to the database in order to improve the overall query evaluation performance. In particular, solutions for the following two problems will be developed:

1. How to dynamically migrate the processing of complex aggregates between the SPE and the database system in response to varying system load and data arrival rate\(^1\) to minimize the query evaluation latency?

2. How to evaluate the migrated continuous queries in the database efficiently by exploiting appropriate data models for streams and query execution strategies?

Note that, computing complex aggregates with huge windows is pervasive in applications. Let us consider the above stock market analysis example: with a 1000 tuples/s data rate, a 24-hour time window will accumulate 3.6 million tuples. One may argue that the window size can be reduced by using stream partitioning techniques [21, 28]. More specifically, the input data stream can be partitioned into several sub-streams based on the business area, and then the correlated aggregation is computed over all sub-streams in parallel. As a result of stream partitioning, the data rate for each sub-stream is lowered, thereby reducing the total number of tuples contained in the 24-hour window. However, in the real-world scenario, there exist tens of thousands of trading records per second even for an individual business area. In this case, stream partitioning cannot be used to reduce the window size because further partitioning the stream will result in incorrect semantics of the computation results. Moreover, according to the Options Price Reporting Authority (OPRA), the estimated peak rates of quotes and trades from options exchanges in January 2013 is about 1 million [34] per second. This implies that, even a window of a few seconds, which are very common for most of stream processing applications, will have several million tuples. Similar use cases can be found in many other application domains such as network monitoring or telecommunications.

In addition, although the motivating experiment results shown in Figure 1 are measured for a specific column-based database system and SPE, the focus of this PhD thesis is to

\(^1\)The data arrival rate influences the size of time-based windows.
explore the superiority of both types of systems to reduce the per-result evaluation latency of complex aggregate queries. The proposed solution is independent of the used database and SPE, as long as they provide the required interface and functionality like the ability of pushing data streams from the SPE to the database.

The remainder of this paper is organized as follows. Section 2 presents related work. In Section 3, we give an overview of the proposed system architecture and explain the functionalities of major components. The solution concepts for two research problems addressed in this paper are described in Section 4 and 5. Section 6 concludes this paper.

## 2. RELATED WORK

The potential of database systems in efficient processing of continuous queries over streaming data has been explored in [19, 23, 10]. Authors in [19] showed that, the performance of stream processing in a standard relational database can be improved significantly by appropriate tunning and use of existing features like indices and temporary tables. The work of [23, 10] presented how to extend the MonetDB and PostgreSQL database systems to support stream processing, respectively. The major motivation of these works is that, by building SPEs separately from database systems, the opportunity of leveraging the existing sophisticated algorithms and techniques of databases is lost. The experimental results of the Linear Road Benchmark [4] given in these works demonstrate that a full-fledged database engine is able to support stream processing. However, their performance results are not compared with existing SPEs. In this thesis, we claim that for stateless operations like filters and stateful operations with small windows, the performance of dedicated SPEs is still superior to these extended database engines. Therefore, a solution which incorporates the virtues of both systems would make more sense to achieve an optimal query evaluation performance.

Botan et al. [7] proposed MaxStream, a federated stream processing architecture which seamlessly integrates multiple SPEs with traditional database. The federation layer is built on top of the SAP MaxDB relational database, acting as the common gateway between client applications and the underlying SPEs and database engines. Data streams and continuous queries are pushed down to specific SPEs through the federation layer. Persistence and joining with static tables is also done in the federation layer if necessary. This work is close to the thesis described in this paper in the sense that it also tried to integrate relatively independent SPEs and database systems. However, in MaxStream, stream processing operations are pushed to the underlying SPEs without considering the possible degradation of their per-result evaluation time. As a result, there is no “migration” and the distribution of actual stream processing is realized on the query level. In contrast, our focus is to identify the most appropriate place for certain complex aggregate operations involved in a query to reduce the overall response time. The query may be very complex and the aggregation operations are only a small part of it. Moreover, due to the varying system load and input data rate, these operations may be migrated between the SPE and the database at runtime.

Recently, a number of general purpose distributed real-time stream processing systems such as S4 [25] and Storm [30] have been proposed, addressing features like scalability and fault-tolerance which are increasingly demanded by modern stream processing applications. Despite the capability of processing streaming data in real-time, these systems also support integrating with external databases as a storage layer. Since one key feature of these general purpose stream processing systems is the support to unstructured streaming data, the databases integrated with these systems are usually nonrelational, now frequently called NoSQL [1], databases. The corresponding terminology for tuple in a relational database is document, object, or extensible record in NoSQL databases [8]. The resulting systems share a similar architecture with the proposed prototype. However, databases in these systems are merely used for the persistence purpose, and there is no built-in mechanisms for leveraging the database to execute tasks with expensive complex aggregate operations. Nevertheless, the idea and method for migrating certain operations from the SPE to the database dynamically is still applicable for these systems, given that the database is more efficient in processing these operations than the SPE. The current focus of this PhD thesis is to combine relational SPEs and databases for stream processing, while the extension to integrated non-relational systems is definitely an interesting research direction in the future.

## 3. PROTOTYPE ARCHITECTURE

To leverage the database system to assist SPEs for processing complex aggregates with huge windows, we envision a prototype system whose architecture is depicted in Figure 2. The core of this prototype system is a component named Migration Manager. Each query coming from the client application is firstly sent to the Dispatcher, which analyzes the client query and detects the existence of complex aggregate operations. For each detected complex aggregation, the Op-Cost Estimator is called to estimate the window size and its per-result-computation-time in both the SPE and the database. Most relational database systems have built-in query cost estimation functionalities, which can be exploited and extended to serve our estimation purpose. Based on the estimation results, the Dispatcher determines at the query deployment time whether it is more efficient to process the operation in the database. Once the migration decisions are
made, the client query is rewritten by the *Query Rewriter* to replace the operations to be migrated with *wrapper operators*. Major functionalities encapsulated in a wrapper operator include:

- Continuously pushing the input data of the migrated operation to the database;
- Triggering the query evaluation in the database at the arrival of new input data;
- Retrieving the computation results from the database, combining them with the partial results computed in the SPE if necessary, and forwarding the results to downstream operators.

The functionality of pushing data from the SPE to the database is not necessarily contained in each wrapper operator. For example, if two operations to be migrated share the same input stream, then data migration only needs to be done by one of the corresponding wrapper operators.

The output of the Migration Manager are modified queries with wrapper operators, which are then deployed in the underlying SPE. However, as a result of the varying system load and input data rate, initially made migration decisions may become inappropriate at runtime. Therefore, the information about the runtime system load, the actual operator performance is collected and forwarded to the *Dispatch* to perform runtime adaptation. Moreover, the *Op-Cost Estimator* also uses this information to adjust the estimation functions so as to improve the estimation quality.

In a nutshell, the goal of the proposed system is to hide all the operation migration details from the end users and transparently leverage the database system to evaluate complex aggregate queries with huge windows in order to reduce the overall query evaluation latency.

## 4. Dynamic Operation Migration

In this section, we sketch the solution approach for the dynamic operation migration problem, including the migration criteria and the fine-grained operation migration strategy. Major challenges involved in solving this problem are also discussed.

### 4.1 Operation Migration Criteria

The key criteria for migrating the processing of a complex aggregate with window size \( w \) from the SPE to the database is to reduce its evaluation latency. The migration decision is influenced by many factors including the per-result-computation-time of the operation in the SPE (\( \rho_{spe} \)) and the database (\( \rho_{db} \)), the data transfer performance between the SPE and the database, and the user-level tolerance on the query response time (QoS constraints). We now discuss these factors one by one.

**Per-result-computation-time.** From a theoretical perspective, the time required to compute one result of the operation depends on the complexity of the operation itself, which is usually a function of the window size \( w \). However, we observed that the actual implementation of operations in a SPE is not always known to the external world, making it difficult to estimate the exact operation complexity. Moreover, the result computation time is also influenced by the currently available computing resources (e.g., CPU, memory) in the physical machine on which the SPE is running.

During system runtime, the amount of available resources is constantly changing due to the varying workload assigned to the machine. The same fact can be observed in the database system. The result computation in the database is done via query evaluation. For a specific query, the evaluation time depends on the chosen query execution strategy and also varies with changing system load. As a result, we believe that statically estimating the per-result-computation-time based on the operator complexity and the system load at the query deployment time only is not good enough. The actual \( \rho_{spe} \) and \( \rho_{db} \) should also be measured dynamically at runtime. How to efficiently get the \( \rho_{spe} \) and \( \rho_{db} \) measurements and how to use these measurements to improve the estimation quality of the *Op-Cost Estimator* will be studied in this PhD work.

**Data Transfer Performance.** We assume that the given SPE provides built-in mechanism to transfer data to the given database, which is called output adapter. The two main performance metrics of an output adapter are throughput and latency. Throughput is defined as the number of tuples sent to the database per second, and latency is defined as the difference between the time when a tuple is received by the adapter and the time when it becomes visible in the database. High data transfer throughput and low transfer latency are two conflicting goals. Therefore, a trade-off between them has to be defined. More specifically, we need to find out that, with the given system setup, (1) what is the throughput threshold (\( \varepsilon \)), below which data of a given complex aggregate operation can be transferred to the database at a near constant latency, and (2) what is the value of that latency (\( l \)). When the input rate of the operation exceeds the detected throughput threshold, a constant data transfer latency cannot be guaranteed. This implies that migrating the operation to the database would not help to reduce the query response time and other solutions have to be resorted. Considering other possible alternatives for this scenario is out of the scope of this PhD thesis and for now, we simply reject the client query.

**QoS Constraints.** Assuming that the input data rate of the target operation is below the detected throughput threshold, then with the measurements of \( \rho_{spe} \), \( \rho_{db} \) and \( l \), we could compare the value of \( \rho_{spe} \) and \( \rho_{db} + l \), and determine whether it is better to migrate the operation to the database. However, before making the final migration decision, the user-defined latency requirements should also be considered. For instance, even if \( \rho_{spe} \) is bigger than \( \rho_{db} + l \), it might be still within the latency range that can be accepted by the user. In this case, it is not necessary to migrate the operation to the database. Another possible situation is that the latency requirement still cannot be satisfied even by migrating the operation to the database. In this case, the query will, again, be rejected.

With the above described migration criteria, we can determine whether a complex aggregate operation in the client query should be migrated from the SPE to the database.

### 4.2 Fine-grained Operation Migration

Migrating one operation from the SPE to the database involves pushing the input data of the operation to the database, as well as triggering query evaluations over these data to calculate the aggregation results. In general, all tuples in the input stream can be pushed down to the database. However, the data movement can also be done at a fine-
grained level, based on the distribution of tuples with respect to specific keys. The notion of keys is similar to that in a relational database. Essentially, a key is an attribute whose value can be used to divide the tuples in a stream into disjoint groups. Each distinct key value represents one data group. Keys can be derived from the group by clause of the aggregation query. For example, the correlated aggregation query introduced in Section 1 shows that the business area attribute can be used as a key of the stock trading data.

The basic idea of the fine-grained operation migration is that, if the computation of the complex aggregation is on the basis of data groups, and the number of tuples in some groups is considerably smaller than that in the other groups, then the tuples of those small groups can be processed within the SPE directly and only the rest tuples need to be pushed to the database. The motivation is based on the fact that the evaluation time of a group by query in the database is also subject to the number of groups within the window. Therefore, by keeping the processing of small data groups in the SPE, we can potentially reduce the query evaluation time in the database. Moreover, the network bandwidth required for transferring these data can also be saved.

However, in real-world scenarios the appearance of different keys normally do not follow a fixed pattern. Therefore, with the sliding of the aggregation window, the distribution of keys within the window is dynamically changing. As a result, the data groups selected to be kept in the SPE must be adjusted dynamically.

In summary, to realize the fine-grained operation migration, we need (1) to collect the runtime key distribution statistics of the migrated streams, and (2) to adaptively change the group-keeping decisions by analyzing the key distribution statistics.

4.3 Other Challenges

The operation migration between a SPE and a database system involves several other conceptual and technological challenges. In the following, we briefly discuss these challenges.

Output Adapter Configuration. To meet variant application level requirements on the throughput and latency, the output adapter provided by the SPE is configurable by a group of parameters. Some parameters emphasize more on the throughput while others emphasize more on the latency. To achieve a best trade-off between the throughput and the latency for a certain data stream under a given system setup, we need to find out an optimal parameter configuration. The procedure of finding the optimal adapter setting for different streams should be automated in the proposed prototype system.

Bandwidth Consumption. The migration of data streams consumes a lot of bandwidth between the SPE and the database system if they reside on different machines. Although in most scenarios, it is valid to assume that the machines running the SPE and the database respectively have very good network connection, the bandwidth is still a potential bottleneck, which must be taken into account when making the migration decisions. When the bandwidth limit is reached, either no more operation migration to the database is allowed, or a distributed database configuration has to be used.

Thrashing. Dynamic operation migration requires reacting to the varying system load and moving the processing of complex aggregates between the SPE and the database adaptively during system runtime. However, under a highly fluctuating system load, a naive adaptive solution can lead to a frequent migration of an operation between the two systems. This issue is called thrashing, which must be avoided because the migration itself is not cost-free.

5. CONTINUOUS QUERIES IN DATABASE

Once some complex aggregate operations are migrated from the SPE to the database, one question that arises immediately is how to efficiently evaluate these operations in the database. This is the second research topic that will be addressed in this thesis. Indeed, this problem can be generalized to the problem of processing continuous query with window-based operations in databases over streaming data, which has been studied by the previous work discussed in Section 2. The two fundamental issues involved in solving this problem is modeling data streams in the database (Section 5.1) and finding a proper query execution strategy (Section 5.2). In this section, we will describe these two issues and revisit the related work by specifically discussing their solutions.

5.1 Model of Data Streams

Streaming data sent to the database has to be kept in certain data structures, upon which operation results can be computed. An appropriate model of data streams is critical to the overall query evaluation performance. It should be storage-efficient and facilitate data sharing between multiple operations processing the same stream. Here, “storage” refers to both memory and disk.

In [10], a window-based operator is implemented through the User Defined Function (UDF), and the required raw data or intermediate results are cached in a data buffer in its function closure. One major problem of this approach is that data buffers are not shared between different UDFs. Thus, even if two operators are processing the same stream, the raw data cannot be shared but has to be replicated in the data buffers of both UDFs. Recall that we are dealing with operations with huge windows. This approach is apparently not storage-efficient. Furthermore, by creating dedicated data buffers, the existing techniques in databases for data management cannot be utilized.

A more intuitive approach is to place the streaming data into database tables. The Continuous Analytics system proposed in [14] transforms a stream into an unbounded sequence of relations/tables based on the window specification in the query. Then the computation is applied to each of these relations. However, the emitted relations overlap one another when dealing with sliding windows, which is very storage-inefficient when the overlap across relations is big. Moreover, the stream-to-relation transformation is done on a per-query basis, which suffers the same data-sharing problem as the solution of [10]. The DataCell architecture described in [23] introduced a new type of tables called baskets to hold a portion of a stream. A basket is a temporary main-memory table, which is constantly updated by adding new tuples and removing old tuples which are already consumed by all relevant continuous queries. Baskets can also be shared by multiple operations. In this case, a tuple is removed from a basket when it is processed by all the operations sharing the basket.

However, the basket maintenance strategy applied in [23]
is too expensive for fast moving streaming data. Therefore, in this thesis, we will adopt the basket concept but relax the tuple removal condition by performing lazy cleanup. That is, processed tuples will not be removed immediately but are kept in the basket until a cleanup threshold is reached. The threshold can be defined based on the time interval or the number of tuples contained in the basket. Meanwhile, correct result semantics is guaranteed by filtering out the out-dated tuples from the computation with proper where clauses.

There exist other optimization opportunities for managing the streaming data. For example, we have the observation that an operation normally does not manipulate all attributes of its input stream. Therefore, the data transfer can be limited to the attributes that the operation is interested in. On one hand, this strategy helps to reduce the amount of bytes to be transferred between the SPE and the database, thereby saving the bandwidth consumption. On the other hand, it saves the storage space in the database. However, when another operation which processes the same stream but requires more attributes is migrated to the database, a method which can gracefully extend the existing basket with more columns, without interrupting the evaluation of existing operations, is needed. In this PhD work, possible optimization opportunities for managing the streaming data in the database will be investigated.

5.2 Query Processing

Given streaming data placed in appropriate data structures in the database, the next problem is how to evaluate continuous queries over these data. The approach applied in [14] is to launch the query repeatedly for each relation produced by the stream-to-relation transformation. In [23], operators of continuous queries are encapsulated in the so-called factories and the execution of factories is organized by a dedicated scheduler. Whenever the execution of a factory is triggered, the partial query plan enclosed in the factory is thrown against its input baskets and the created result set is then placed in its output baskets. Authors of [10] proposed a cut-and-rewind query execution model, in which a data stream is cut into a sequence of chunks and the query is evaluated over each chunk sequentially. The main difference to [14] and [23] is that the query instance never shuts down between chunks, thereby avoiding the effort of re-parsing, re-planning and re-initializing the query.

In this PhD work, the performance of the above mentioned query evaluation methods will be compared and the most efficient approach will be selected. Additionally, we also want to explore the power of existing features in the database to improve the query evaluation efficiency. For example, materialized views and, in particular, their self-maintenance techniques [18, 26] can be applied to speed up the calculation of the inner aggregate of a correlated aggregation operation. Another widely used and powerful technique is indices, whose main purpose is to speed up data accessing.

6. CONCLUSIONS

Motivated by the observations that (1) the computation of complex aggregates with huge windows is pervasive in a large application domain, and (2) a modern database system is able to evaluate these operations more efficiently than existing SPEs when exact answers are demanded, the PhD thesis presented in this paper proposes a prototype system, which leverages modern database systems to assist SPEs to evaluate complex aggregate queries by dynamically migrating the operation from the SPE to the database. This involves studying two core problems: (1) how to realize meaningful and dynamic operation migration under varying system load, and (2) how to evaluate the migrated operations in the database efficiently. In this paper, a prototype architecture is proposed and solution concepts for the two problems are presented. These solutions will be implemented as key components of the Migration Manager, which is the core of the proposed prototype system.

7. REFERENCES


