Triton: A Continuous Query Translation Engine for Trident/Storm

Zhiheng Li
Department of Computer Science and Engineering
University of California, San Diego, CA 92122
Email: lzhiheng@eng.ucsd.edu

***Abstract*—In this paper, we present Triton, a system which takes a script written in a continuous query language *TQL*, and produces compilable and readable native JAVA code that runs directly on the Storm, a distributed real-time computation platform. The Triton system is designed to be flexible so that multiple target programming language generators can be implemented since Storm is also a polyglot environment. The Triton system acts the same role on Storm in the real-time processing field as the famous *Pig* system to *Hadoop* in the batch processing area. In addition, several implementations of the sliding window in Storm are discussed and compared.**

I. Introduction

A. Background

A fairly large amount of researches have been done in the area of continuous query processing and complex event analysis since early 2000. The *STREAM* prototype Data Stream Management System (DSMS) designed by the Stanford InfoLab [2] is marked as one of the pioneer projects in this area. The *CQL* language, a continuous query language designed along with *STREAM* DSMS defines an abstract semantics for the *sliding window*, which is a key component in streaming analysis, and thus became an important reference when researchers design new query languages for streaming based systems. Base on these work, several Complex Event Processing (CEP) engines and Event Processing Language (EPL) have been proposed to ease the development of real-time streaming applications. Esper [10] is one of the successful examples that enables a SQL-standard, low-latency and real-time streaming CEP engine. However, one problem that these systems suffer is the scalability and reliability since most of them perform in-memory computation. The only way to scale systems for massive data sets is adding more physical memories. On the other hand, the single-machine architecture becomes another bottleneck when streaming rate increases. With the increasing needs in application areas such as financial analysis, network monitoring, online bidding and algorithmic trading, a distributed real-time computation system designed with horizontal scalability and fault-tolerant model that produce high throughput and low latency results is often desired. Apache Storm [12], a distributed real-time computation system designed by Nathan Marz, has been adopted by several big companies like Twitter, Groupon, and RocketFuel since its first release around early 2009. It provides a fault-tolerant, scalable and at-most-once semantic model according to its official website. Apache Storm introduced Trident, a high level abstraction over Storm, in its 0.8.0 version released on 08/02/2012. [13] The advent of Trident made the distributed real-time computation even simpler since it comes with a Domain Specific Language (DSL) abstraction that allows programmers focus on the data operations only during the development. However, there is no *sliding window* primitives existed in Trident, and a higher level abstraction that provides a SQL-standard language for streaming analysis is still missing.

B. Motivation

The motivation of our work is partially inspired from the *Pig* project [8] done by the Yahoo! research, who proposed a novel language called *Pig Latin* for performing map-reduce job. The *Pig Latin* program will be translated into low-level physical plans that runs directly on *Hadoop*, an open-source implementation of map-reduce framework. [**?**] As claimed in the documentation that Storm can be viewed as a real-time version of map-reduce framework, our goal is to design and implement a system which acts as the similar role over Storm as *Pig* did over *Hadoop*. The system is named as *Triton*, which is a translation engine that compiles a script written in *Triton Query Language* (TQL) into native JAVA code (in form of a Trident program) that runs directly over Storm. *TQL* is a SQL-standard continuous query language with

syntax similar to *CQL* and Esper. In summary, we made three major contributions in this project. 1) *TQL* was developed as the query language for the *Triton* system. 2) An experimental implementation of the *Triton* translation engine that compiles the *TQL* query into native Trident program written in JAVA. 3) Several implementations of the *sliding window* primitive in Trident was investigated.

The rest of this paper is organized as follows. In the next section, we will talk about the key concepts in Storm and Trident to give readers an overview of the platform that our system depends on. In section III, we will cover the design of *TQL* and provide several examples. In section IV, we will introduce the system architecture, design decisions and the rationale behind them. In section V, we will describe the current implementation of the Triton system. Section VI will present one real world application that is implemented through the Triton system. We will discuss the project experiences in section VII. Finally, we will list the future work and related work in section VIII and IX, then make the conclusion.

II. Storm and Trident Overview

In this section, we will quickly go through two core concepts, *stream* and *topology*, in Storm, and give a brief introduction to Trident. More detailed descriptions and tutorials about the system can be found in their official documentation. [12]

A. Stream

Stream, the core abstraction in Storm, is defined as an unbounded sequence of tuples, where each *tuple* is an ordered list of objects with pre-defined data types. A Stream can be transformed into a new stream by two Storm primitives *spout* and *bolt*.

1) Spout: A spout is defined as the source of a stream. A spout can turn the data from a queue or database into a stream. Storm provides an interface for users to implement application specific spouts.

2) Bolt: A bolt is a functional unit that takes arbitrary number of streams as input, and emit tuples as a new stream based on the logic defined by the application. Technically a bolt can do any kind of computation such as filtering, aggregation, projection and so on. Storm also provides an interface for application specific implementation. A complicate computation often involves multiple bolts.

B. Topology

A topology in Storm is a directed acyclic graph (DAG) where each node is either a spout or a bolt.

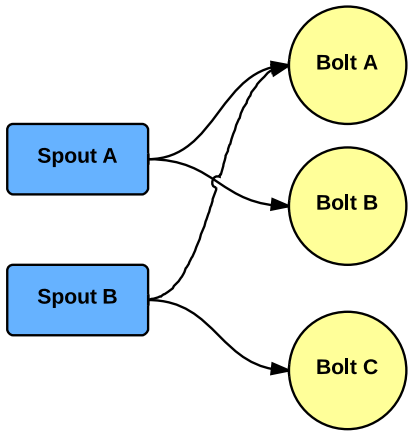


Fig. 1. An example of a Storm topology.

More precisely, spouts can only have out degrees, acting as a source of the topology, while bolts can be put anywhere in the topology. Each concrete computation task is modeled as a topology and run either on a storm cluster or locally. Figure 1 shows a simple topology that contains two spouts and three bolts. Notice that spout A emits stream and send it to three bolts C, D and E, and bolt D received stream from two spouts A and B. Figure 2 demonstrate a concrete example.

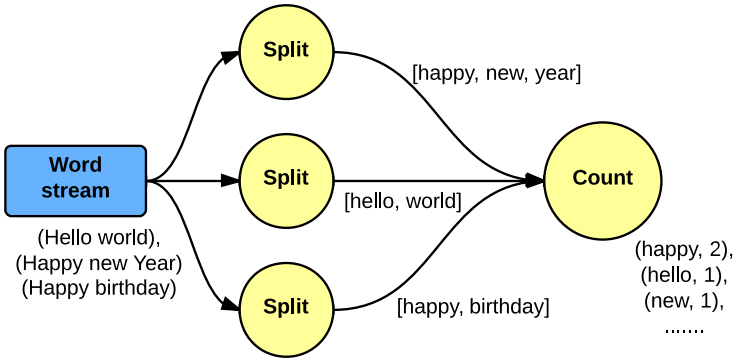


Fig. 2. A Storm topology that takes a word stream and appends two exclamations to it.

C. Trident abstraction

Trident is built on top of Storm, aimed to provide a high-level abstraction for doing real-time computing. Common operations on stream including functions, filters, aggregations, and projections are wrapped as Trident APIs. The details of how these operations are implemented in terms of Storm topologies are hidden from

users. In addition, Trident's API is designed as a fluent interface to produce more readable code. The Trident engine will automatically translate a Trident topology into a Storm topology. A more detailed description of Trident fluent API will be covered in Section V-E.

III. Triton Query Language

A *TQL script* consists of a sequences of *statments*, with each of them terminated by a semicolon. Each statement is either a *stream registration* or a *query definition*. Each stream must be registered into the system before being referenced in a query. Section III-A and III-B cover the detail.

A. Stream registration

A stream registration begins with a `REGISTER STREAM` statement, followed by the stream name and the attribute definition list. Each attribute definition in the list contains the attribute name and the type. There are four kinds of type our system supports currently, integer, float, string and timestamps. The user also need to specify a source definition indicates where the stream should read data from. The source definition can be a user-defined spout class such as `spout("BatchSpout")` or a disk file such as `file("input.data")`. Example III.1 shows a stream registration reading from file data.

Example III.1.

```
REGISTER STREAM s(id   integer
                  word string,
                  time timestamps)
FROM file("word.dat");
```

B. Query definition

The query definition in *TQL* is pretty similar to the standard SQL, with additional syntax extension for sliding windows and data output. Our system supports three kinds of sliding windows, row-based, time-based, and time batch sliding window. Each window definition can has a pre-filter on it, which means each tuple that does not meet the filter conditions will not go into the sliding window. Example III.2 defines a one minute sliding window contains stock information about Goolge only.

Example III.2.

```
stock[code='GOOG'].win:time(1 minute)
```

For each query, we can specify an `OUTPUT` clause that indicates where the results of the query should go to. The destination is a user-defined function, and can be database, file, network, and so on. If there is no output specified, the result will be directed to `stdout` by default.

A query can also be registered as a derived stream by specifying the query as a source in the stream registration. Example III.3 shows an example of a derived word count stream, where the result of the word count query is registered as a new stream named `wordCountStream` into the system. We can also view the derived stream as a *named query* since the query has a name (the stream name), and can be referenced by other queries.

Example III.3.

```
REGISTER STREAM
wordCountStream(word string,
                wordCount int)
FROM
  SELECT word, count(word) as wordCount
  FROM wordStream.win:time(1 minute)
  GROUP BY word;
```

As far as this paper is written, our system supports common aggregation functions such as `COUNT`, `SUM`, `AVG`, `MIN` and `MAX`. We also allow simple arithmetic expressions appears in `SELECT` and `WHERE` clauses. Stream and attribute renaming can be enabled by the syntax `<name> AS <new_name>`. `ORDER BY` and `LIMIT` is also included. There is no support for set operations such as `UNION` and nested query yet.

IV. System Design

A. Design decisions

1) System architecture: From a high level of view, the system should be designed in a modularized way, with each module has a clean interface exposed to communicate with each other. Modules should be relatively independent from each other so that it can be easily replaced by a different implementations as long as the interfaces remain the same. This leads to a flexible and extensible system architecture, where new module can be added and old module can be replace without too much code change. Another advantage is that several implementation of a same module could be compared so that the optimal one can be used in the system. This design also leads to a generic code generation phase, where we aimed to provide a generic interface that allows code generator in different programming languages such as Python or Ruby can be written. This design decision cater to the flavor of Storm since itself is also language independent platform. The core of Storm is written in Cloujre, but the exposed APIs can be written with different programming languages.

2) Compilable and readable code generation: Regarding to the code generation, the system should produce compilable JAVA code that contain equivalent semantics to the high level *TQL* scripts, and can be run directly on the Storm topology. This implies several implementation challenges such as variable name allocation, import package management, and code organization. In addition, the generated code should be readable. This requirement can be view from two aspects. First it indicates that the code is pretty printed with proper indents between each logic blocks. In this case, we designed an internal representation of a JAVA program, which is a tree data structure, and a language printer which can print the program in a formatted way. Readable code also implies the logic of the code should be easy to understand. To achieve this, our system will translate the query script into Trident program, and delegate the translation of Storm topology to the platform itself. As mentioned above, the fluent interface ensures the program is clean and readable. However, there are several drawbacks of this design. First, the efficiency of the generated Storm topology may be affected due to the two-stage translation, though Trident engine will perform several optimizations during the process of topology generation. Second, this design leads to a high level implementation of the sliding window, which again may have a negative impact on the performance. Intuitively, it will be faster if this primitive is embedded in the Storm core system.

These design goals will serve as important guidelines when we design the system architecture and should be followed to a large extent.

B. System overview

The high level system architecture is show in Figure /refarch-diag. The input of our system is a *TQL* script where the stream registration and query are specified. The output is a compilable JAVA program that can run on the Storm to execute the desired queries. The entire compilation involves five major stages: parser, logic query plan generator, optimizer, translator and JAVA code generator. In addition, there is a resource manager that servers as a meta-data manager during the compilation. The global information about the input query is stored and updated in the resource manager. For example, the stream registration will store the stream definition into resource manager. All the renames and stream dependencies information are kept in resource manager too.

The parser takes a *TQL* script as an input, and parsed it into a abstract syntax tree (AST). In the meantime, the parser collected all the necessary information such as attribute names and types and store them into corresponding nodes for further translation. The logic query plan generator takes the AST as input and produce a logic query plan by traversing the AST. The logic plan is then scanned by the optimizer to explore the possibility of optimization. Currently two kinds of optimization is implemented, join detection and selection push-down. The optimized query plan will be passed into the translator, where the translation is performed on each logic operator in the query plan to produce a Trident program. In the last stage, the trident program will be passed into the JAVA program generator, where the necessary packages, classes and files will be generated and saved into a user-specified folder.

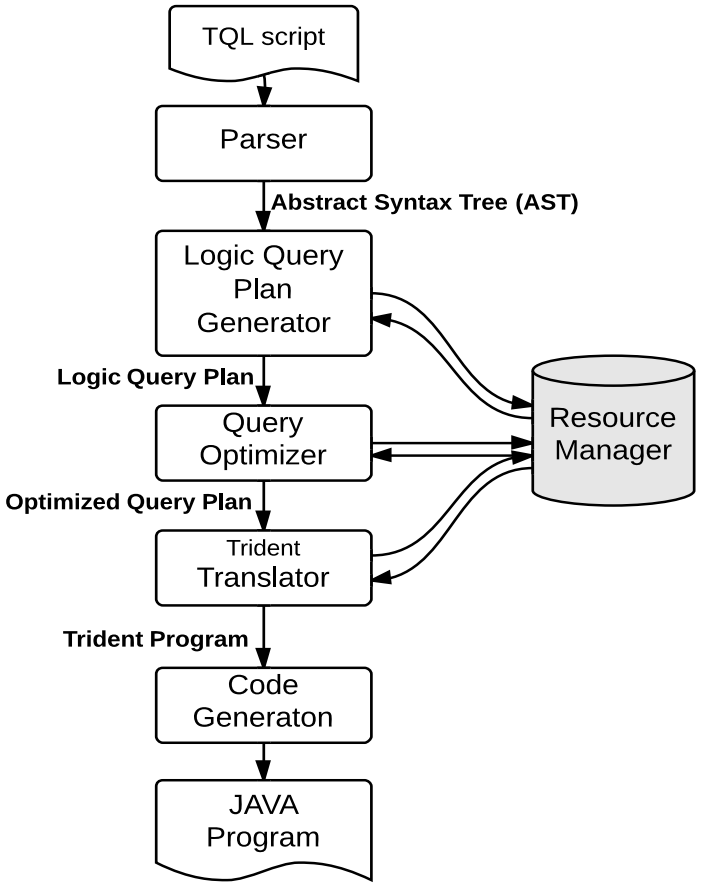


Fig. 3. System architecture diagram.

V. Implementation Details

A. Parser

The first stage of the compilation is to parse the input *TQL* script and generate the abstract syntax tree (AST) for further translation. This step is simple by using a handful tool called JJTree, a powerful extension of JavaCC to perform lexical, syntactical analysis, and construct abstract syntax tree during the parsing process. We only need to specify the BNF grammar in a `jjt` file and pass it to JJTree, and JJTree will generate a parser written in JAVA. Two things need to be taken care of when writing the `jjt` file, one is the left-recursion elimination to avoid the parser error and the other is the usage of `LOOKAHEAD` function to eliminate the possible ambiguity in the grammar.

B. Meta-data management

During the compilation process, one key component is the management of *meta-data* of a given script. In a *TQL* script, the meta-data includes the stream definitions (stream name, attribute/type and source), stream/attribute renaming information, query type(named or anonymous), and query dependencies information. All these information will be collected and stored in proper data structures during the logic query plan generation phase.

`ResourceManager` and `LogicQueryPlan` are two major data structures involved in the meta-data management component. `ResourceManager` is a singleton class that maintains a table of stream definitions, which is internally represented as a hash table keyed by stream name. `ResourceManager` will be only instantiated once in the run-time. The will be collected when processing the `REGISTER` node.

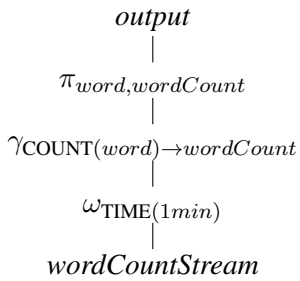
`LogicQueryPlan` contains the meta-data of each query, including stream/attribute renaming information, query type(named or anonymous), and query dependencies information. The system creates one `LogicQueryPlan` object for each query in the script. The renaming is captured when analyze the `AS` node, and stored in a hash table where key is the new stream name and value is the old stream name. The query type information is actually an flag variable stored in the `LogicQueryPlan` object. The dependencies for each query are stored as a linked list, which will be used to construct a dependency graph in the code generation phase.

C. Logic Query Plan Generation

The logic query plan in TQL is similar to the one in standard SQL. To incorporate the sliding window operation, we extend the relational algebra by introducing a window operator ω. In addition, if the window definition contains a pre-filter, a selection operator σ should be placed before ω. For instance, the relational algebra for the window specification in Example III.2 can be written as $\omega_{TIME(1min)}(\sigma_{symbol='\text{GOOG}'})$.

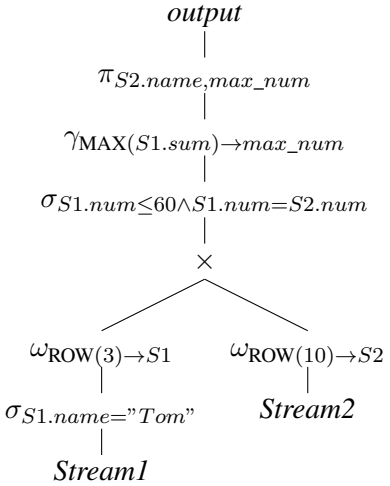
The translation from input query to a logic query plan is simple in our cases since there is no nest query supported in the system. Example V.1 demonstrate a simple logic query plans generated by our system.

Example V.1. A simple logic query plan.



The implementation of the logic query plan generation is done by traversing the AST produced by the parser. During the traversal, depending on what kind of nodes, new operators will be created or existing operators will be updated. For example, a σ operator will be created when a `WHERE` node is visited, and will be updated when each of its condition nodes is visited. When the traversal is completed, all the necessary operators should have been created, and will be organized in a specific way. A product operator will be placed before input streams if there is more than one streams. The generated logic query plan is also stored in the `LogicQueryPlan` as a tree data structure. A logic query plan involves two streams is shown as below.

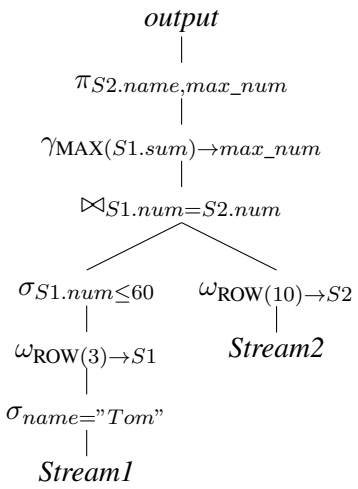
Example V.2. A logic query plan involves two streams.



D. Optimization

Two optimization techniques were adopt when possible to produce better logic query plan, join detection and selection push-down. Both of the optimizations were implemented by analyzing the logic expression of the `WHERE` clause. A pre-processing decomposes the logic expression into an equivalent expression which contains list of logic units joined by $\wedge$ whenever possible. In case if there is no such decomposition, neither of the optimizations can be done. This decomposition is useful because each small unit is independent of each other and can be processed separately because of the $\wedge$ logic. Example V.3 shows a optimized version of Example V.2. The logic expression $S1.num \leq 60 \wedge S1.num = S2.num$ is decomposed into $S1.num \leq 60$ and $S1.num = S2.num$, where $S1.num \leq 60$ is pushed down to the *Stream1*, and the product between *Stream1* and *Stream2* is converted to a join operation based on $S1.num = S2.num$.

Example V.3. An optimized logic query plan.



1) Selection push-down: Selection push-down is relatively easy based on the decomposition. The condition that a logic unit can be pushed down is that if it only contains attributes from the same stream, which can be verified through a traversal on the logic unit.

2) Join detection: For join detection, a linear scan was done on each logic unit to check if it is a equal expression and the attributes in two sides belong to different streams. If this is the case, a `JoinPlan` object will be created, and the joined stream pair will be added into it. After the scan, a join graph will be generated where each node is a stream and each edge between two node means a join exists between them. The join attribute is stored as a linked list in the edge. Then a graph partition algorithm will be applied to find the join groups, which is simply a depth-first-search. At this time, join operator could be placed on each join group. If there are more than two streams in a join group, a left-deep join tree will be generated by default. Currently we have not investigated the cost-based plan generation yet.

E. Translation and Code Generation

1) Trident API overview: Before discuss the technique for translating *TQL* to Trident program, we first look at what does a typical Trident program look like. Example V.4 shows a code snippet of the implementation of a word count application. The code is pretty self-explained if you familiar with JAVA.

Example V.4.

```
/* A split function that takes string as input
 * and split it by space.
 */
public static class Split extends BaseFunction {
  @Override
  public void execute(TridentTuple tuple,
      TridentCollector collector) {

    String sentence = tuple.getString(0);
    for (String word : sentence.split(" ")) {
      collector.emit(new Values(word));
    }
}

topology.newStream("spout1", spout)
  .each(new Fields("sentence"), // input fields
        new Split(),   // user-defined function
        new Fields("word"))    // output fields
  .groupBy(new Fields("word"))      // group by
  .persistentAggregate(
    new MemoryMapState.Factory(),
    new Count(),                  // aggregation
    new Fields("count")        // output fileds
  .parallelismHint(16);
```

- `newStream` takes a user-defined spout object as input and specify a stream id.
- `each` processes each tuple emitted by spout and apply the user-defined function on it to emit new fields. These new fields will be appended to the original one. A filter can also be plugged in the each function.
- `groupBy` groups all the tuples with the same group by fields will be grouped into same partition.
- `persistentAggregation` runs an aggregation function over the global stream and store the result into `TridentState`, where a `TridentState` can be anything like database or in-memory storage.

The following APIs are also useful and therefore are listed.

- `project` emits the fields specified in the project input.
- `partitionPersistent`
- `aggregation` runs aggregation function on each partition.
- `applyAssembly` applies a user-defined function to manipulate the total stream, for example, sort.

2) Translation of ω (window operator): The window operator translation is implemented by three Trident APIs. As illustrated in Example V.5, a `s1.win:time(1 min)` window could be expressed by a sequence of `partitionPersistent`, `newValuesStream` and `groupBy`. The idea is that each time the spout emit a tuple, it goes into the `TimeWindow`, which is a buffer ring internally. The buffer ring supports a policy-based eviction specified by the Window class. For row-based sliding window with size n, the eviction policy is all the tuples that are n away of the current one should be eliminated. For time-based sliding window with duration d, the policy is all the tuples whose timestamps satisfies $|t_{old} - now| > d$ should be eliminated. The `SlidingWindowUpdater` will maintain the sliding window based on the eviction policy, and emit all the tuples in current window with a unique windowId each time when a new tuple enter the window. The output stream can be captured by the `newValueStream`, and `groupBy` makes all the tuples with same `windowId` go to the same partitions.

Example V.5.

```
_topology.newStream("s1", _spout)
  .partitionPersistent(
    new TimeWindow.Factory(60),
    new Fields("word"),
    new SlidingWindowUpdater(),
    new Fields("windowId", "word"))
  .newValuesStream()
  .groupBy("windowId")
```

3) Translation of other operators: The translation of other operators such as σ and π are relatively simple.

- Selection can be translated through `each` API, where the condition clause will be translated into a class derived from `BaseFunction`.
- Projection can be translated through `project` directly if no arithmetic expressions appears in the `SELECT` clause. If there exists arithmetic expression, an `each` and its function class will be generated as we did in the selection operator.
- Aggregation is translated through `aggregate` directly. For multiple aggregations, the translation should begin with a `chainAgg` and end with a `chainEnd`. For the implementation of `AVG`, we use two chained aggregator `COUNT` and `SUM`.
- Order by and limit is tranlsate to the `applyAssembly(fisrtN())`, where fisrtN is a merge-sort based stream assembler.

Table I summarized the required APIs of each operator translation.

TABLE I
A MAPPING BETWEEN LOGIC OPERATOR AND TRIDENT API

Logic Operator	Trident API
Selection	each with Filter class
Projection	each with BaseFunction class project
InputStream	newStream
OutputStream	each
Aggregation	groupby aggregation persistentAggregation
Window	partitionPersistent newValueStream groupby
Join	join
OrderBy	applyAssembly(firstN)

4) Code Generation: One issue in code generation phase is the variable naming for the function class required by the `each` API, the output field of an expression or aggregator if no output field is specified. To avoid ambiguity among queries. Each name is prefixed by the query plan name, which is assigned uniquely through the script.

Since the query can be specified in any order in the script, the generated query plan list may not contains correct order for code generation. A topological sort must be applied on the logic query plan list to produce an correct order for sequence of plans based on the query dependencies information stored in each query plan.

The import package management is done by maintaining a static import list which covers all the packaged that a query may refer to. We plan to support dynamic import list generation in the future. '

In the last phase of the code generation, Triton will create a folder for the generated project, and generate a default MAVEN script for building. The JAVA code will be stored into a file specified by the client. The system will also copy all the related classes into the project to minimized dependency.

VI. Example

A. Trending Topic

Trending topic is defined as a word, phase or topic that is tagged at a greater rate than any other words. In other words, the trending topic is a measurement of what is hot now, and it can tell the users what is happening in the world currently. In the following example, we assume that the data we use is pulled from the public Twitter API.

```
# This is a sample query script that produce
# the trending topic of a word stream.

# register a word stream
REGISTER STREAM wordStream(word string)
FROM file("data/word.dat");

# compute word count for past 1 min.
REGISTER STREAM
wordCountStream(word string, wordCount int)
FROM
    SELECT word, count(word) as wordCount
    FROM wordStream.win:time(1 minute)
    GROUP BY word;

# compute top 10 word
SELECT word FROM wordCountStream
ORDER BY wordCount DESC LIMIT 10;
```

VII. Project Experience

Typically, a master project lasts from one to three quarters, requiring a long-term commitment for students. I have learned some valuable experiences during the development of project. First of all, starting early will be a good indication of a successful project. It is not uncommon that a master project requires additional technical backgrounds or platforms that students were unfamiliar with before. Getting started with these backgrounds makes students well prepared for the actual projects. Second, a long-term project requires a clear delivery goal being kept in mind all the time, so that students will not get lost in the middle of the project. Regarding to the development and coding, since design decisions can always be modified during the development, keep the interface clean and flexible can largely reduce the amount of work on code refactoring. In addition, package management and build automation tools such as Maven and Ant can ease the pain of maintaining large amount of external dependencies and compilation issues, and are necessary for project development and maintenance.

VIII. Future Work

A. Query Plan optimization

In the current implementation of the *TQL* query plan generation phase, only two kinds of query optimization techniques were adopted, join detection and selection push down. The produced query plan could be further improved if multiple join operators were detected. Our system will generate a left-deep join tree by default if multiple join presents. One way to improve the query plan is to introduce a cost-based plan generation using the techniques in the area of traditional relational database management system. **??**. On the other hand, the existing optimizations were implemented directly in the query plan generation phase so that it is hard to add new optimizations. Our next step is to build a rule-based optimization layer on top of the query plan to seperate the query plan optimization from query plan generation.

B. Built-in sliding window support

One of the bottleneck that will affect the performance of the Triton system in a significant way is that there is no primitive sliding window operation in the current release of Trident. The way we simulate the sliding window utilized some existing APIs, and it may not be the optimal solution. However, it is possible to implement the sliding window feature directly in the core storm system, and expose it to trident for our translation.

C. User Defined Function

It is desired that our system could become more flexible and extensible if User Defined Function (UDF) is supported. For example, if user-defined Java code is allowed in the script, customized filter functions can be plugged in the WHERE clause, function for data transformation can be done in the SELECT clauses, and a lot more. To achieve this goal, we need to defined the proper UDF interface, extend our parser to accept the Java syntax, and then perform the translation.

D. Performance benchmark on cluster

Due to the time and resource limit, only local tests have been done in this project for now. However, it is not sufficient for a distributed real-time system to have local test only. To fully evaluate the performance of our system, a series of benchmark performance experiments should be done in a cluster environment with large set of real data. In this way, we could monitor the actual delay, the failure-tolerant behavior, the average latency and throughput and so on.

IX. Related Work

A. STREAM

STREAM, the Stanford Data Stream Management System, is one of the pioneer project leaded by the

Stanford InfoLab database group that proposed a general purpose data stream-management system. It defined a formal abstract semantics for the continuous queries and designed the *CQL* language to implement these concepts. The *CQL* extends the classic *SQL* syntax to support sliding window and stream-relation conversion. There are three types of sliding window implemented in the STREAM system, row-based, time-based and partition-based. The system is run in memory. When the data rates exceeds its ability to provide accurate result, a *load-shedding* algorithm was applied to provide approximate results.

B. Esper

Esper is a powerful Complex Event Processing (CEP) engine, which provides a large set of functionality for CEP jobs. Esper offers a Domain Specific Language (DSL) for processing stream events. Also, Esper provides a declarative language, the Event Processing Language (EPL), for processing high frequency time-based event data. *SQL* streaming analytic is another commonly used term for this technology. One disadvantage of the system is about scalability. The Esper system runs entirely in memory, so the only way to scale the system is to add more physical memories to it. In addition, the system suffers the single-point failure issue since the system is deployed on a single machine.

C. Trident-Esper

Trident-Esper is one attempt that people made trying to integrate the trident and esper so that the advantages of both systems could be retained. The approach they use is to turn the entire computation of the Esper Engine into a bolt in the trident. They created a new bolt which wrapped up the Esper operations inside it. They provided the interface for the esper to have the ability of reading input stream directly from the spout, and emit result into another node in the trident topology. Though this integration requires only a few lines of code change and seems elegent, one major issue is that the all the computation is done in in-memory and not fully parallel.

D. Pig-Latin

Pig Latin is a novel data processing language developed by the Yahoo! Research. It combines the feature of high-level declarative querying in *SQL* and the low-level procedural programming paradigm map reduce. The program written in Pig Latin can be compiled into a set of map-reduce job that executed on the Hadoop system to perform data analysis jobs. It has been widely adopted by many companies since its first release and become one of the necessary component during the deployment of a big data application. The idea and the design of the Triton project is also partially derived from the Pig Latin. To some extent, we could claim that Triton is the *Pig Latin* for Trident.

X. Conclusion

We presented a continuous query translation engine base on the Trident/Storm platform called Triton, and a query language called *TQL* along with the Triton system. The goal of Triton is to provide a high-level abstraction on Trident to reduce the difficulty of building a distributed streaming processing application. The *TQL* script will be compiled into native Trident program in JAVA programming language by Triton, and run directly on the Storm platform. Users no longer need to worry about too much low level details such as how to construct a Trident topology and how to scale the system.

Currently Triton is still in its early stage, but we believe that when it becomes mature, it will play an important role in the area of real-time streaming processing, just as what *Pig* has achieved.

Acknowledgments

I would like to thank Professor Amarnath Gupta for his great intuition and advice on this project, and Nathan Marz, the father of Storm and Trident, for his innovative work on creating such an amazing platform for distributed real-time computation. I will not be able to accomplish this project without the support of their work.

References

[1] Arasu, A., Babu, S., and Widom, J. (2006). The CQL continuous query language: semantic foundations and query execution. *The VLDB JournalThe International Journal on Very Large Data Bases*, 15(2), 121-142.

[2] Arasu, A., Babcork, B., Babu, S., Datar, M., and et al. (2003, June). STREAM: the stanford stream data manager (demonstration description). *In Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (pp. 665-665). ACM.

[3] Motwani, R., Widom, J., Arasu, A., and et al. (2003, January). *Query processing, resource management, and approximation in a data stream management system*. CIDR.

[4] Srivastava, U., and Widom, J. (2004, June). Flexible time management in data stream systems. *In Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (pp. 263-274). ACM.

[5] Abadi, D. J., Carney, D., Ċetintemel, U., et al. (2003). Aurora: a new model and architecture for data stream management. *The VLDB JournalThe International Journal on Very Large Data Bases*, 12(2), 120-139.

[6] Arasu, A., Cherniack M., Galvez E., et al. Linear Road: A Stream Data Management Benchmark. *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, August, 2004.

[7] Krämer, J., and Seeger, B. (2009). Semantics and implementation of continuous sliding window queries over data streams. *ACM Transactions on Database Systems (TODS)*, 34(1), 4.

[8] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. (2008, June). Pig latin: a not-so-foreign language for data processing. *In Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (pp. 1099-1110). ACM.

[9] Paul Dekkers, "Complex Event Processing," M.S. thesis, Dept. Computer Science, Radboud Univ., Nijmegen, October 2007.

[10] Paul Jensen, "Complex Event Processing with Esper" [Online]. Available: http://jnb.ociweb.com/jnb/jnbOct2008.html

[11] Schätzle, A., Przyjaciel-Zablocki, M., and Lausen, G. (2011, June). PigSPARQL: Mapping sparql to pig latin. *In Proceedings of the International Workshop on Semantic Web Information Management* (p. 4). ACM.

[12] Storm project [Online].
Available: http://storm.incubator.apache.org/

[13] Trident Tutorial [Online].
Available: https://github.com/nathanmarz/storm/wiki/Trident-tutorial

APPENDIX

A list of core BNF syntax for *TQL*

```
<query> ::= <select_clause> <from_clause> [<where_clause>]
            [<group_by_clause>] [<order_by_clause>] [<limit_clause>]

<select_clause> ::= select <select_list>

<from_clause> ::= from <stream_def> (, <stream_def>)*

<where_clause> ::= where <filter_criteria>

<group_by_clause> ::= group by <expression> (, <expression>)*

<order_by_clause> ::= order by <attribute> (asc | desc) (, <attribute> (asc | desc))*

<select_list> ::= *
                |  <select_attribute> (, <select_attribute>)*

<select_attribute> ::= <attribute> [as <identifier>]
                     | <expression> [as <identifier>]
                     | <aggregate_function> (<attribute>) [as <identifier>]

<stream_def> ::= <stream_name> [(<filter_criteria>)] [. <view_spec>] [as <identifier>]

<filter_criteria> ::= <filter_criteria> and <filter_criteria>
                    |  <filter_criteria> or <filter_criteria>
                    |  not <filter_criteria>
                    |  ( filter_criteria )
                    |  <expression> <cmp_op> (<expression> | <constant>)

<expression> ::= <arithmetic_expression>

<attribute> ::= <identifier>
              |  <attribute> . <identifier>

<view_spec> ::= win:length(number)
              |  win:time(time_period)
              |  win:time_batch(time_period)

<time_period> := <num> <time_unit>

<time_unit> := hr | min | sec

<cmp_op> ::= >
           | <
           | >=
           | <=
           | =
           | <>

<aggregate_function> ::= count | avg | sum | median | stddev

<identifier> ::= [a-zA-Z][0-9a-zA-Z_]*

<constant> ::= <number_literal>
             | <string_literal>

<number_literal> ::= <integer> | <floating>
```