Technical Report
1199

Knowledge Query Language (KQL)

S.K. Damodaran

1 February 2016

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LEXINGTON, MASSACHUSETTS

This material is based upon work supported by
the Assistant Secretary of Defense for Research and Engineering (ASD R&E)
under Air Force Contract No. FA8721-05-C-0002 and FA8702-15-D-0001.

This report is the result of studies performed at Lincoln Laboratory, a federally funded research and development center operated by Massachusetts Institute of Technology. This material is based on work supported by the Assistant Secretary of Defense for Research and Engineering (ASD R&E) under Air Force Contracts No. FA8721-05-C-0002 and FA8702-15-D-0001. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of ASD R&E.

Massachusetts Institute of Technology
Lincoln Laboratory

Knowledge Query Language (KQL)

S.K. Damodaran
Group 59

Technical Report 1199

1 February 2016

Lexington Massachusetts

This page intentionally left blank.

EXECUTIVE SUMMARY

Currently, queries for data retrieval from non-Structured Query Language (NoSQL) data stores are tightly coupled to the specific implementation of the data store implementation, making portability of the queries or query-dependent algorithms difficult. This report introduces an ontological declarative approach that is independent of the storage content and format for querying NoSQL or relational data stores. This approach uses *address expressions* (or A-Expressions) embedded in commonly used query languages such as Structured Query Language (SQL). The declarative approach makes the queries portable, and results in several advantages over the existing approaches to querying, especially when the data is semi-structured, and when the data sources may change over time. Cyber event logs are examples of such data sources. When the query is independent of the underlying physical data sources, having provenance information on the query results becomes important to impart necessary context, and ensure trust in the query results returned. This declarative approach is made possible through the use of a knowledge registry. In this report, we discuss embedding A-Expressions in the widely used SQL, resolving A-Expressions using the ontology implemented in a knowledge registry, and returning query results with provenance information.

This page intentionally left blank.

ACKNOWLEDGMENTS

The author would like to express his gratitude to Alexia Schulz for support during the development of A-Expression and its exposition, Tamara Yu for initial collaboration on knowledge registry in the context of LLCySA project, David O'Gwynn for implementing an early version of the knowledge registry in the tool he developed, and Pedro Colon-Hernandez, for completing a reference implementation of KQL queries over a MongoDB data store while he was an intern. The author would also like to thank Stephanie Mosely and Jerry Sniegocki for help with the editing of this document.

This page intentionally left blank.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

Page

LIST OF FIGURES

This page intentionally left blank.

1. INTRODUCTION

An example of a modern distributed key/value store is Google's BigTable [1]. Big Table is best described as a sparse, distributed multidimensional sorted map. Unlike a relational database, BigTable has no multicolumn primary keys or constraints. The lack of a table schema works well when storing and retrieving unstructured data, such as documents. However, when semi-structured data such as event logs are stored in key/value stores, the row-key of a table is used to retrieve data in *string* format from the key/value store. Currently, queries for retrieval are tightly coupled to the specific implementation of the key/value store implementation, making portability of the queries or query-dependent algorithms difficult. The querying problem becomes much more complex when the data sources from which data is collected are subject to change, as in the case of cyber data sources. For example, new data sources may be added with only some of the fields of another data source log record. An approach for querying that is independent of the storage content and format becomes necessary under these circumstances. Also, when the query is independent of the underlying physical data sources, having provenance information on the query results becomes important to ensure trust in the query results returned.

We describe in this report address expressions, or *A-Expressions*, for storage-independent addressing of information stored in a data store, embedding A-Expressions in the widely used Structured Query Language (SQL), resolving A-Expressions using an ontology, and returning query results with provenance information. We describe in detail the mechanisms used by A-Expressions to resolve to columns and tables in a data store. The address expression may be used in ad hoc queries or embedded queries to retrieve contents from a key/value data store. Our addressing scheme offers several benefits stemming from the independence of the addressing scheme from the storage content and format.

This page intentionally left blank.

2. DETAILED DESCRIPTION

Figure 1 describes a simplified view of traditional query processing using data stores [2]. In circled step 1, a query is passed from an Analytics Platform to a Query Parser, which processes the query, and then passes the parsed query down to a Query Executor (step 2). Query Executor executes the queries over a data store or a distributed platform of data stores (step 3), and returns the results to the Analytics Platform (step 4).

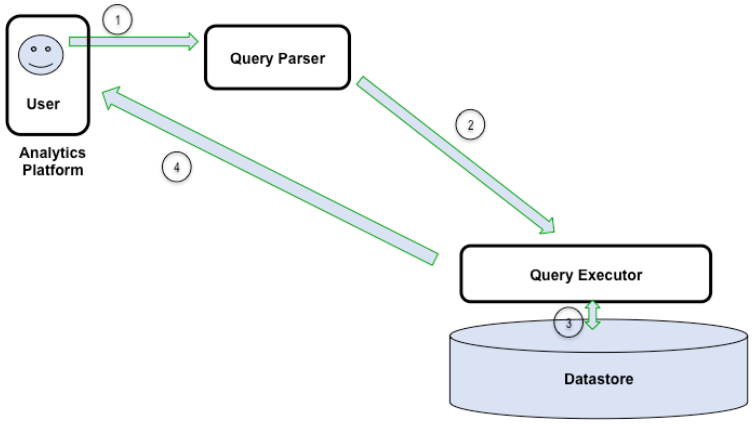

Figure 1. Traditional query processing.

The query languages used in these queries can vary, and range from variations of SQL to specialized languages such as PIG [3] and HIVE [4]. Often, custom program snippets in programming languages such as Python that directly refer to the physical names of the underlying data store elements are passed to the Query Parser, which in turn passes these to the Query Executor for execution over the data store. This approach is very powerful but requires a highly skilled user who is well versed in the physical data store implementation content and format.

An improved approach using a knowledge registry was proposed in [5]. A knowledge registry is maintained by a knowledge engineer. In Figure 2, we show a pictorial summary of this approach. In circled step 1, the query is passed to the Query Analyzer by the user directly, or through an Analytic Application. In step 2, the query, specified in a custom query language, is analyzed using the contents of the knowledge registry by Query Analyzer. This analysis consists of mapping the ontology elements in the query to physical data store elements such as columns and column families in the data store. In step 3, the mapped query is converted into executable program snippets by the Query Executor, and executed over the data store. In step 5, the results of the execution are returned to the user.

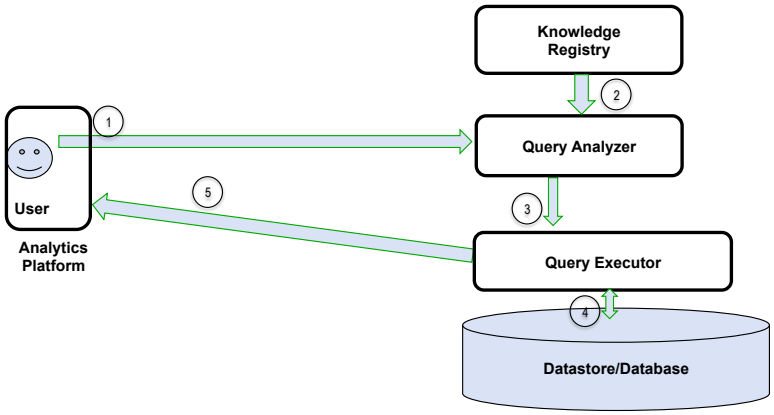

Figure 2. Knowledge registry–based querying.

In Figure 3, we describe an improved approach to what is depicted in Figure 2. The key aspect of the improved approach is that the query structure itself is declarative, as opposed to program snippets, and uses address expressions, or A-Expressions. Such expressions are also embedded in query languages such as SQL.

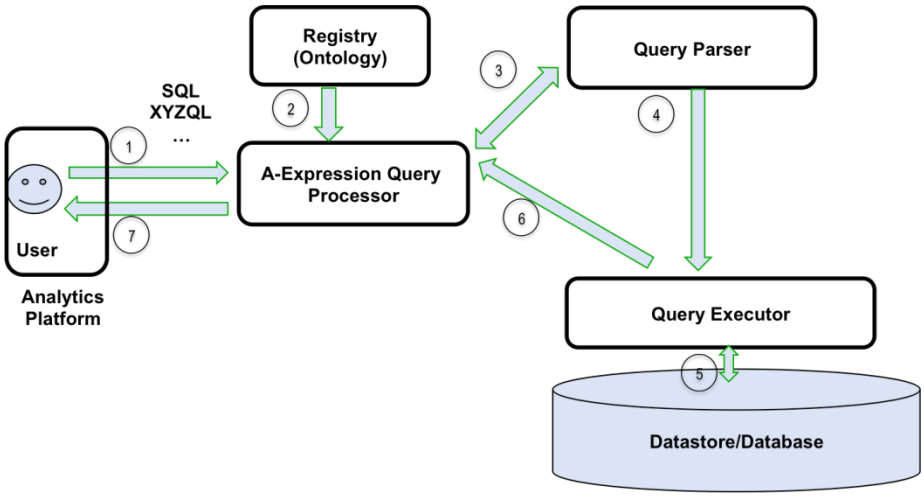

Figure 3. Address expression-based querying.

Each circled step in Figure 3 is described below.

1. User submits a query with embedded address expressions to A-Expression Query Processor (AQP).

2. AQP extracts the embedded A-Expressions from the query, and evaluates them using the ontology stored in the registry, and records provenance information returned from the registry.

3. AQP generates an SQL query based on A-Expression evaluation, and submits to Query Parser.

4. Query Parser parses the query and passes to Query Executor.

5. Query Executor executes the query and returns the results or error(s) if the query cannot be executed.

6. AQP receives the results, combines with provenance information.

7. AQP returns the results of the query with previously recorded provenance information to the user for use in analytic application.

Figure 4, below, describes the internal processing within A-Expression Query Processor (AQP) when some of the processing in Figure 3 occurs. Each numbered item in Figure 4 corresponds to a circled processing step in Figure 3.

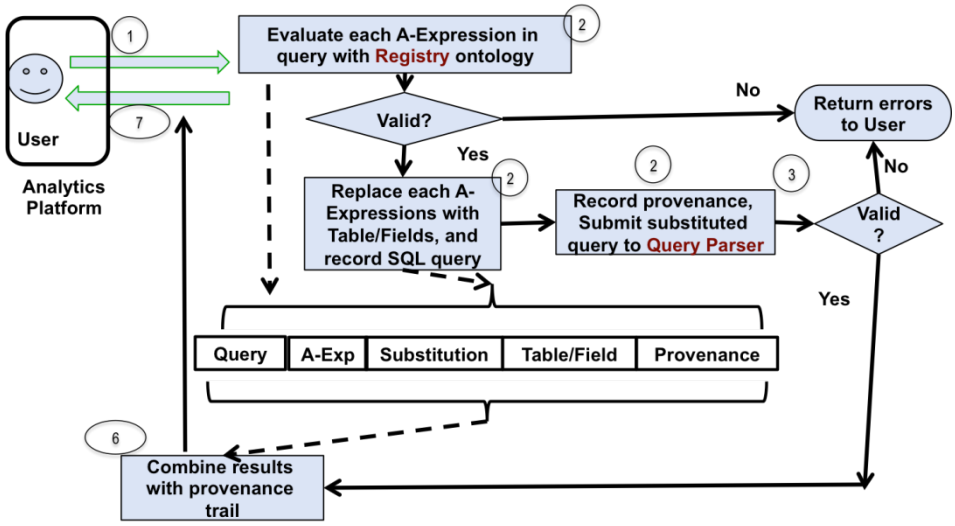

Figure 4. Internal processing of AQP.

1. A query in a query language such as SQL with embedded A-Expressions is submitted by the user through a Query/Analytics Platform.

2. The A-Expression Query Processor (AQP) checks each of the A-Expressions in the query for syntactical and semantic validity. If there is any error, then AQP returns errors to the user. AQP assigns to each A-Expression a unique identifier string, and records the A-Expression, the corresponding query, tables and fields that would be substituted for the A-Expression in the original query based on A-Expression evaluation using registry ontology, a unique identifier for each such substitution or rewriting, and a provenance trail for the evaluation, as shown in Figure 4. These activities correspond to Step 2 in Figure 3.

3. Rewrite the SQL query with resolved tables and fields from A-Expression evaluation with registry (Step 2 in Figure 3), and submit the rewritten version of the query, which is a syntactically correct SQL query, to a SQL query parser (Step 3 in Figure 3).

4. Parser parses the query and query is processed with Query Executor (not shown in Figure 4, but corresponds to Step 4 in Figure 3).

5. Query Executor executes the query over the data store, and returns the query results to AQP (not shown in Figure 4, but corresponds to Step 5 in Figure 3).

6. AQP combines the results and the provenance trail for each A-Expression in the query (Step 6 in Figure 3), and presents to the user (Step 7 in Figure 3).

In the following sections we will describe in detail how the processing steps described in Figure 3 and Figure 4 are implemented. In particular, we describe the registry ontology, its implementation, a description of A-Expression and its evaluation over the ontology, and examples of embedding A-Expressions in SQL queries.

2.1 REGISTRY ONTOLOGY

Figure 5 describes the registry ontology to support AQP processing described in Figure 3. We describe here dimension, dimension set, operator, field, and table schema described in [6] that continue to be used for AQP processing. We also define new ontology elements, tag, tag scheme, virtual dimension, derived dimension set, and enumerated dimension that will be used to do AQP processing. We describe below each of the ontology elements that are relevant to AQP processing in detail, including some of the constraints. Instances of this registry ontology are created for all the data sources that are ingested. These instances are used for query analysis. We collectively call the instances and the ontology schema described in Figure 5 as *registry ontology* for convenience. In some implementations, the instances of registry ontology are embedded in other components such as AQP, and they may not be explicitly stated. In fact, the implementation we describe in later sections stores the instance of registry ontology in JSON files.

Throughout the rest of this report, we use the term *field* to refer to the physical column of a data store. A field content is part of a row of storage in the data store. Since our area of application of these technologies are for log file processing, a row of the data store corresponds to a log file record, and a field corresponds to an element within the log, such as *userid* or *hostname*. We use the term table to describe a collection of log records or their subsets. One or more column families can be considered tables since a log record may be stored into a single column family or multiple ones, based on performance considerations.

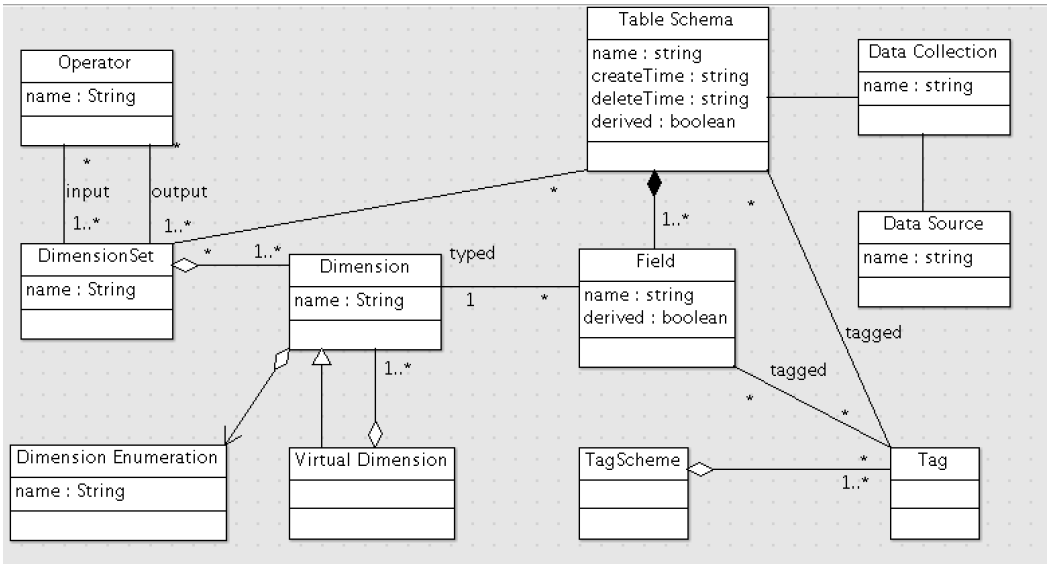

Figure 5. Registry ontology.

2.2 DIMENSION

Every field may be assigned a type, an immutable attribute of the field, which may have a specific semantic interpretation or syntactic structure sensible for the domain in consideration. For example, *IPAddress* is a type in cyber log files. We refer to this type as *dimension* to avoid confusion with data types in programming languages. The *type* is used to identify and interpret a column entry. The dimension of a field is assigned by a user, more specifically by a knowledge engineer, and this mapping between fields and dimensions is stored in the knowledge registry. In key/value store implementations, in the absence of any user-assigned dimension, a field may have a single default data type of *string*. If the data is stored in relational databases, there may be additional options for default types, such as *integer*, assigned to the fields. These data types are not considered dimensions.

Every field typically can have at most one dimension, and once assigned by the user, it is not changed, though a field may be reinterpreted as a *virtual dimension* (see below). Multiple fields from the

same or different tables can have the same dimension. For example, in Figure 6, *Netflow.Field1* has dimension *Protocol,* while *Netflow.Field1* and *Proxy.Field4* have dimension *IPAddress*.

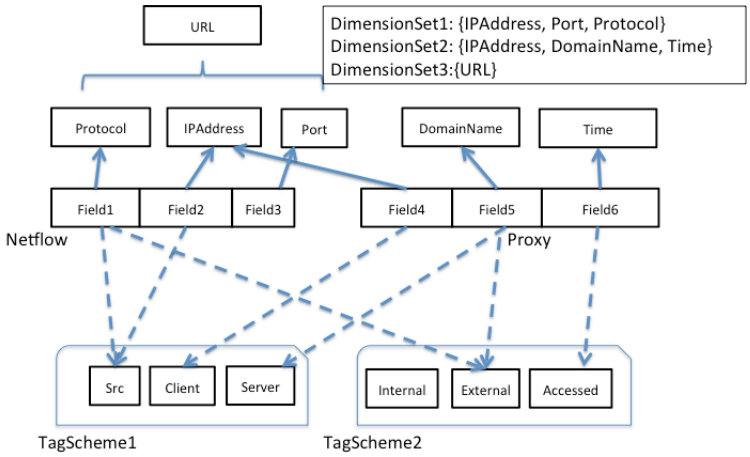

Figure 6. A concrete example.

2.3 DIMENSION ENUMERATION

A dimension has an implied or explicit range of values that it can assume. For example, an IPv4Address has a well-defined range of possible values. When the range of values a dimension can assume needs to be explicitly enumerated, a *dimension enumeration* is used to define all such values. For example, a field may have an ACTION dimension, and only two values (SEND or RECEIVE) can be valid in this field. This means, if a field has a dimension of ACTION, there are two enumerations, SEND and RECEIVE.

2.4 VIRTUAL DIMENSION

Often it becomes necessary to reinterpret the contents of a field with additional dimensions. To address this situation, even though a field may already have an assigned dimension, we use the concept of *virtual dimension* to address this requirement. Dimensions can be aggregated to another dimension that we call *virtual dimension*. For example, in Figure 6, the URL aggregates to <*Protocol*, *IPAddress*, *Port*>. This aggregation may be the result of a requirement to assign to a set of dimensions a sequential order. The sequential ordering specified in virtual dimension is useful when parsing the contents of a new data source, and parts of the data source content may be interpreted as one or more dimensions. Aggregation or virtualization of a dimension may also occur due to reinterpretation of the content of a field into additional dimensions at a later time. For example, in Figure 6, the URL field may previously had a dimension *URL* that maps to a single field currently consisting of *Field1*, *Field2*, and *Field3* in the Netflow table. It is possible that a need to break the content of the field into *Protocol*, *IPAddress*, and

Port, may have arisen subsequently, resulting in *URL* becoming the virtual dimension consisting of these fields. However, all fields corresponding to dimensions in a virtual dimension must be in the same table.

The same dimension may be part of multiple virtual dimensions. For example, in Figure 6, the field sequence <*Protocol*, *IPAddress*, *Port*> is a virtual dimension, *URL,* while <*IPAddress*, *Port*> can be another virtual dimension, such as *BindingAddress.*

2.5 DIMENSION SET

Dimensions may be grouped together in a set without implying any sequential ordering among the dimensions, in contrast to virtual dimensions. Such grouping is supported by the concept of dimension sets. The concept of dimension set is useful for doing queries without regard to where its constituent dimensions are stored. For example, in Figure 6, *DimensionSet1* has dimensions of *IPAddress*, *Port*, and *Protocol* with no implied order or physical contiguity of data corresponding to these dimensions. Note that a dimension set may be defined corresponding to a virtual dimension, if needed.

As another example, consider how in Figure 8, *DimensionSet1* has two dimensions, *Dimension1* and *Dimension2*. *DimensionSet2* consists of *Dimension1* and *Dimension3*. Note that since *Field4* and *Field5* have the same dimension, *Dimension3*, *Dimension3* is only included once in *DimensionSet2*.

A dimension set does not need to correspond to any existing table. Dimensions in a dimension set also need not correspond to dimensions of any existing fields in a table, though it may be convenient to do so in the early stages of a development of a knowledge registry for a data store. Instead of making dimension sets map to the existing fields in tables, the users can specify dimension sets that would make sense from the point of view of the user who specifies queries within a specific domain. In addition, the term *DimensionSet* itself may be different in different domains. For example, in the domain of *Cyber Events*, these dimension sets may be called *Events*. *House Listing* and *Apartment Listing* may be dimension sets in the domain of *Real Estate*.

2.6 TAGS AND TAG SCHEMES

Tags have been widely used as a means of categorizing and retrieving unstructured data. Personal tags allow categorizing data in terms meaningful to a person. A *tag* is a keyword, or qualifier, assigned to a piece of information. A tag is a kind of metadata that helps describe an item and allows it to be found again by browsing, searching, or querying. A dimension is analogous to a noun, whereas a tag is analogous to an adjective. Tags may be chosen informally by the item's creator or by its viewer, or by the knowledge engineer, depending on the system. Tags may also be standardized for a set of data items. One key aspect of tags is that the same item may have multiple tags. A specific user may know of only a subset of these tags. Tags may also be organized in tag schemes. Tag schemes may be created by individual users to avoid conflicts with tag schemes created by others, and then shared with others, or knowledge engineers who may create some standard tag schemes for an organization. Tags within a tag scheme may have relationships among them, or have no relationships. A common organization of a tag is

a *tag cloud* where the tags do not have any relationship with each other. Equivalence relationships may be defined between individual tags, belonging to a single tag scheme or multiple tag schemes, allowing for substitution of one tag for another in an A-Expression.

A field or table can be assigned tags from one or more tag schemes. When there are no tag schemes defined in a particular implementation, we assume all tags belong to the same default tag scheme. The same field or table may be assigned multiple tags from a single or multiple tag schemes. A tag scheme has a set of tags that may have arbitrarily relationships among them. For example, in Figure 6, *TagScheme1:Src* is a tag for *Netflow.Field1* and *Netflow.Field2*. *Netflow.Field1* and *Proxy.Field5* have *TagScheme2:External* as the tag.

2.7 DIMENSION SET REACHABILITY THROUGH DATA OPERATORS

A dimension set can also be represented as a function of one or more dimension sets and a *data operator* that operates on the values of the specified dimension sets and/or scalar values. As an example, consider a COUNT operator that counts the entries with a specific dimension in a field. Note that these data operators operate on the values stored in tables and fields. In key/value stores, data in dimension sets are often *fused* to create another dimension set. This fusion operation is another example of the aforementioned data operator. The new *DimensionSet X* that came into existence due to a data operation on another *DimensionSet Y* is referred to as *derived from DimensionSet Y*. A dimension set may also be derived from multiple dimension sets.

The relationship among input dimension sets and output dimension sets for an operator are stored in the knowledge registry. The input and output relationships are used to infer the *derived from* relationship between dimension sets through the specific data operator.

An example of derivation relationships is shown in Figure 7, where *DimensionSet13* is derived from *DimensionSet15* and *DimensionSet14*. A derived dimension set is semantically equivalent to a regular dimension set, and is treated as such in A-Expressions.

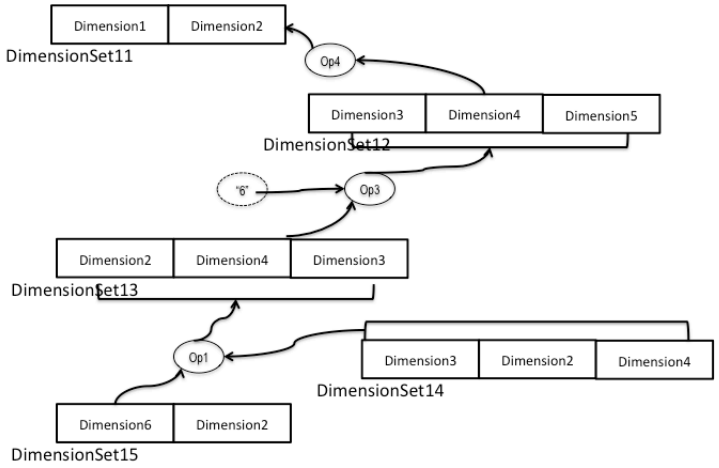

Figure 7. Derived dimensions.

Figure 7 shows a set of dimension sets related through a set of operators. In Figure 7, all except *DimensionSet15* and *Dimension14* are derived dimension sets. The information on which operators led to the creation of the *DimensionSet11* from *DimensionSet14* and *DimensionSet15* will be stored in the knowledge registry. It is always possible that any of the dimension sets in Figure 3 could refer to tables that obtained data directly from sensors or data that was derived from other dimension sets and not a derived dimension set at all. Therefore, it is important to be able to clearly identify the derived data fields and tables as distinct from non-derived fields and tables. In Figure 5, the field and table classes have an attribute, *derived,* that can be true or false. The derived data is stored in the data store, and the knowledge registry is updated with appropriate metadata related to the derived data. This report does not address the confidentiality or privacy concerns that arise from such storage.

2.8 ADDRESS EXPRESSIONS OR A-EXPRESSIONS

In a distributed key/value store, data is stored in tables or column families, and can be retrieved with keys. The tables have columns, which we refer to as *fields* in this report. The fields are location dependent in two ways: (1) every field has a position that is unchangeable within a table in the data store, and (2) a field can be referenced only with respect to a table because a field is an integral part of a table. This location dependency makes it very hard to implement generalized queries and ad hoc queries that only refer to the field names or the table names. This report will define multiple ways to define storage-independent addresses for the fields, and tables, and how to resolve such addresses to fields in one or more tables. Queries may use these addresses, which we term A-Expressions, to specify one or more fields or tables. The resolution of the A-Expression to a set of fields is done over a knowledge registry. We described an architecture to this resolution is earlier sections. In the rest of this report, we focus on the

details of how such resolution is done. While this report focuses on evaluating A-Expressions over key/value data stores, the described techniques are not specific to key/value stores. A-Expressions can be used effectively over other types of data stores such as relational databases, as long as a knowledge registry is also implemented.

A-Expressions such as *IPAddress,* where *IPAddress* is the name of a field, are used to specify all the data within a field. Queries for a specific subset of data within a field require in addition to the specification of fields, expressions such as *IPAddress=173.1.3.1.* In such expressions, any reference to a field in a table or a reference to a table in any query can be replaced by an A-Expression. However, the semantics of operators such as = are defined in the query language in which A-Expressions are embedded; for example, SQL. We discuss examples of using A-Expressions within SQL later in this report.

2.9 COMPONENTS OF A-EXPRESSIONS AND THEIR EVALUATION

An A-Expression may be constructed using the following types of components: dimension, dimensionset, tag, tag scheme, and a set of registry operators defined for these components. We refer to these operators as *registry operators* because these operators cannot be evaluated over the data in the data store but only over the ontology stored in the knowledge registry. We describe an example of these registry operators later in this report. An A-Expression, when evaluated over the data schema in a knowledge registry, yields a set of tables or fields. An A-Expression does not contain any direct reference to any table or field. In the subsequent paragraphs, we define these components, registry operators, multiple examples of A-Expressions, an example implementation of a registry ontology instance, and an example implementation of the registry operators.

Fields and tags use the syntax of the form *Table.Field* and *TagScheme.Tag* respectively to denote the fact that a field can only be defined relative to a table. similarly, a tag does not have an independent existence without the tag scheme. The character _ denotes a default tag scheme. *ALL* refers to all tables, or all fields in the registry depending on the registry operator context. Below are the registry operators used in A-Expression.

1. The */* registry operator is used to operate on tables and returns tables containing all dimensions specified as a dimension set, or as a set of dimensions.

2. The * registry operator is used to operate on tables or fields and returns fields matching dimensions or tags.

3. The . is a registry operator that operates on fields and returns fields that existed specific durations specified within *{* and *}*.

4. *(* and *)* are used to describe to impose an evaluation order of A-Expression in that the A-Expression within a parenthesis will be evaluated first prior to what is outside.

5. NOT (!), AND (&), and OR (|) are logical registry operators. The NOT operator is a universal negation.

6. *{* and *}* is used to create sets of dimensions, dimension sets, tags, or durations using a *,* separation.

7. *[* and *]* is used to extract the set of dimensions in a set of dimension sets.

8. The *?* is a registry operator that operates on a dimension set and returns all unique reachable dimension sets from the given dimension set through data operators.

See Appendix A for the full set of ANTLR [7] based rules for parsing A-Expressions.

2.10 EXAMPLES OF A-EXPRESSIONS

2.10.1 Examples for Dimensions Based on Figure 8

An A-Expression, *ALL * Dimension1,* will resolve to all fields with that dimension. In Figure 8, *Dimension1* will resolve to *EventTable1.Field1* and *EventTable2.Field3*.

Contrast this example with a situation where there are no dimensions defined. In this situation, *Field1* and *Field3* need to be referenced explicitly as *EventTable1.Field1* and *EventTable2.Field3*, respectively. Every referenced field in *EventTable1* and *EventTable2* will need to be known to the analyst. Describing in terms of dimensions allows specification of both of these fields, or any not yet existing field in a table with the dimension of *Dimension1*.

Let us say that a new *EventTable3* with a *Field34* with *Dimension1* is added in the future. The same A-Expression *Dimension1* would resolve to *EventTable1.Field1*, *EventTable2.Field3*, and *EventTable3.Field34*. Thus, the user who specifies the expressions does not need to know anything about the availability or physical location of fields in the tables.

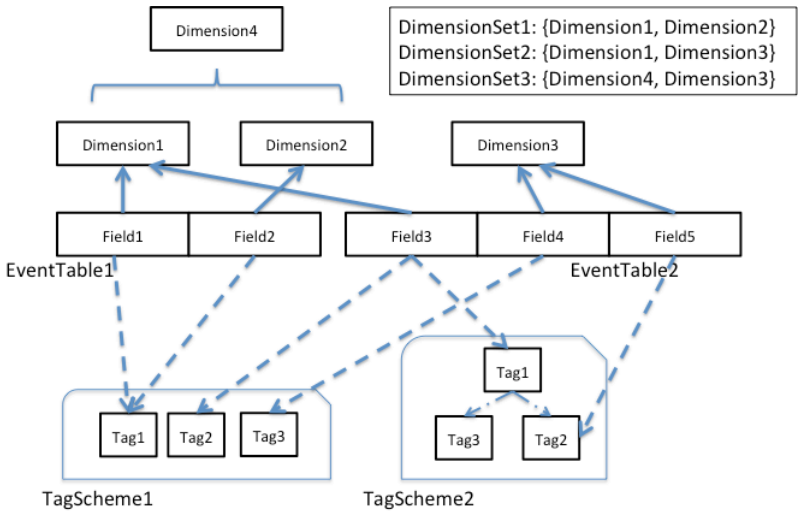

Figure 8. Table, field, dimension, dimension set, tag, tag scheme.

2.10.2 Virtual Dimension Examples Based on Figure 8

A virtual dimension is processed in A-Expressions just like a dimension. As a consequence, the dimensions that are part of a virtual dimension will only resolve to fields that are within a single table, and these dimensions map to adjacent fields in the exact sequence in which they are defined in the virtual dimension.

An A-Expression of *ALL * Dimension4* would resolve to fields consisting of *Dimension1* and *Dimension2*, {*EventTable1.Field1*, *EventTable1.Field2*}, but not to {*EventTable2.Field3*, *EventTable1.Field2*}.

2.10.3 Dimension Set Examples Based on Figure 8

A dimension set can be used in A-Expressions, as a short cut for specifying all the dimensions in that dimension set individually. Such an expression resolves to tables that contain all the fields corresponding to the dimensions in the dimension set. For example, *DimensionSet1* may be used to identify *EventTable1* because only *EventTable1* has all of the dimensions of *DimensionSet1*.

Furthermore, a dimension set can also be used to narrowly specify a particular field with a specific dimension. For example, *ALL/DimensionSet1 * Dimension1* will resolve to exactly *EventTable1.Field1*, and *ALL/DimensionSet2 * Dimension1* will resolve to exactly *EventTable2.Field3* in Figure 1.

In A-Expressions, instead of an already defined dimension set, an anonymous dimension set may be defined as a set of dimensions. For example, *ALL/{Dimension1,Dimension3}* is equivalent to *ALL/DimensionSet2*.

It is also possible to use *[* and *]* operators to specify the dimensions in a set of dimension sets. For example, *ALL * [DimensionSet1] * Dimension1* will resolve to *EventTable1.Field1* and *EventTable2.Field3* in Figure 8. The *[* and *]* set of operators is useful when the name(s) of the dimension set(s) are known but not the dimensions within them.

2.10.4 Tag and Tag Scheme Examples Based on Figure 8

Tags and tag schemes also provide an alternate and powerful way to unambiguously specify a subset of fields resolved with an A-Expression consisting of only dimensions and dimension sets. For example, in Figure 8, an A-Expression of *ALL * Dimension3* can resolve to both *EventTable2.Field4* and *EventTable2.Field5*. Therefore, we need a way to unambiguously specify just one of those fields using an A-Expression. We will describe below how to use tags in A-Expressions to do just this.

A tag scheme has a set of tags which may have arbitrarily relationships among them. For example, all the tags in a tagging scheme may be related, as in *TagScheme2*, or not related at all, as in *TagScheme1*, or somewhere in between.

The tag schemes may be created and shared by knowledge engineers, or analysts (end users), and stored in the registry or elsewhere, as described in Figure 3 and Figure 4.

2.10.5 Reachability-Based Examples

In Figure 7, the A-Expression *DimensionSet13?* will resolve to {*DimensionSet12*, *DimensionSet11*}.

2.10.6 Duration-Based Examples

A-Expression *ALL/DimensionSet1 {2013-05-30T09:00:00, 2013-05-30T10:00:00} * Dimension1* will return *EventTable1.Field1* if it exists or has values within that duration.

2.11 COMPOSING A-EXPRESSIONS

The expressiveness of A-Expressions come from the way A-Expressions may be combined. Below we give different examples of these combinations.

2.11.1 Composing A-Expressions with Dimensions, Dimension Sets, and Tags

In Figure 8, an A-Expression *ALL/DimensionSet2 * Dimension3 * TagScheme1:Tag3* will resolve exactly to *EventTable2.Field4*. An A-Expression of *ALL/DimensionSet2 * Dimension3 * TagScheme2:Tag2* will resolve to *EventTable2.Field5*.

In Figure 8, A-Expression *ALL/DimensionSet2 * TagScheme1:Tag2* will resolve to *EventTable2.Field3*, whereas *ALL * Dimension2 * TagScheme2:Tag1* will resolve to *EventTable2.Field3* as well. However, note that two different tag schemes were used. It is possible that *TagScheme1* is created by *KnowledgeEngineer1* and *TagScheme2* is created by *Analyst1* based on their individual ideas on how the information in the data store must be interpreted.

Sometimes, a user may not know of existing dimenson sets, but would know the tags. Sometimes, dimension sets may not have been defined, yet tags are assigned to fields. In such cases, using logical operators within an A-Expression containing tags from one or more tag schemes may be used. Such A-Expressions can be used to make the resolution more precise.

2.11.2 Composing A-Expressions with Logical Operators

In Figure 8, *ALL * Dimension1 * (TagScheme1:Tag1 & (! TagScheme2:Tag1))* will resolve to *EventTable1.Field1*. If the user knew of *DimensionSet1*, an equivalent A-Expression for *Field1* would be *ALL/DimensionSet1 * Dimension1 * TagScheme1:Tag1*.

In Figure 8, *ALL/DimensionSet2 * (([DimensionSet2]) & (! (Dimension1)))* will return all fields in *EventTable2* except ones with *Dimension1* but with dimensions specified in *DimensionSet2*. The returned fields would be *EventTable2.Field4*, and *EventTable2.Field5*.

It is quite possible that a dimension set may resolve to multiple tables. In such cases, tags may be used to distinguish them. For example, *(ALL/DimensionSet1) & TagScheme1:someTag* could be used to identify only tables with *DimensionSet1* with the specified tag.

2.12 IMPLEMENTATION OF A-EXPRESSION ONTOLOGY

There are two key elements to resolving A-Expressions over the knowledge registry: (1) storing the A-Expression components, the relationships among them and with tables, and fields in the knowledge registry ontology, and (2) algorithms to resolve the A-Expressions using the stored ontology in the knowledge registry. We describe below an instance of the stored relationships in knowledge registry, and the associated algorithms in the next section. Note that this is only an example of implementation using JSON format, and these entities and relationships can be stored in other formats such as XML, RDF, or OWL.

Table to Field Map: This map specifies the fields in each table. The map corresponding to Figure 8 is below in JSON format.

```
{
"tables":
[
   {"name" : "EventTable1", fields": [ {"name": "f1"},{"name":"f2"} ] },
   {"name" : "EventTable2","fields" : [ {"name": "f3"},{"name":"f4"},{"name":"f5"}] }
]
}
```

Field to Dimension Map: This map describes the dimension corresponding to each field. Below is the map for Figure 8.

```
{
"f2dmap":
   [
            {"table" : "EventTable1","field": "f1","dimension": "d1"},
            {"table" : "EventTable1","field": "f2",dimension": "d2"},
            {"table" : "EventTable2","field": "f3","dimension": "d1"},
            {"table" : "EventTable2","field": "f4","dimension": "d3"},
            {"table" : "EventTable2","field": "f5","dimension": "d3"}
            ]
}
```

Dimension Set to Dimensions: This map describes all the dimensions in a dimension set. Below is a map for Figure 8.

```
{
"dimensionSets":
  [
            {"name": "ds1","dimensions": [ {"name": "d1"},{"name":"d2"} ]},
            {"name": "ds2","dimensions" : [ {"name": "d1"},{"name":"d3"}]},
            {"name": "ds3","dimensions" : [ {"name": "d4"},{"name":"d3"}]}
  ]
}
```

Virtual Dimensions to Dimensions: This map describes the dimensions that are part of a virtual dimension. Below is the map for Figure 8.

```
{
"virtualDimensions":
   [
            {"name": "d4","virtualDimensions": [ {"name": "d1", "position": "1"},{"name":"d2", "position": "2"} ] }
   ]
}
```

Tag Scheme to Tags Map: This map describes all the tags within a tag scheme. Below is an example for Figure 1.

```
{
"tagSchemes" :
   [{"name": "ts1","tags": [ {"name": "t1"},{"name":"t2"}, {"name":"t3"} ] } ,
   {"name": "ts2","tags" : [ {"name": "t1"}, {"name":"t2"}, {"name":"t3"}] }
   ]
}
```

Tag to Field Map: This map describes the fields that a tag is associated with. Below is an example for Figure 8.

```
{
"tag2fieldItems":
   [
            { "table" : "EventTable1", "field": "f1", "tagScheme": "ts1","tag": "t1"},
            { "table" : "EventTable1","field": "f2", "tagScheme": "ts1","tag": "t1"},
            { "table" : "EventTable2","field": "f3","tagScheme": "ts1", "tag": "t2"},
            { "table" : "EventTable2","field": "f3","tagScheme": "ts2", "tag": "t1"},

            { "table" : "EventTable2","field": "f4","tagScheme": "ts1","tag": "t3"},
            { "table" : "EventTable2","field": "f5","tagScheme": "ts2","tag": "t2"}
   ]
}
```

Data operator to input dimension sets or input dimensions; output dimension set or output dimensions: This map describes the “in” and “out” parameters to each operator. Below is an example based on Figure 7.

```
{
"operators": [
   {"name": "op1","dimensionSetIn": [ {"name": "ds15"},
                                      { "name": "ds14"}
                                     ],
                    "dimensionSetOut": {"name": "ds13"}},
   {"name": "op3","dimensionSetIn": [{"name": "ds13"}],
                   "dimensionSetOut": {"name": "ds12"}},
   {"name": "op4","dimensionSetIn": [{"name": "ds12"}],
                    "dimensionSetOut": {"name": "ds11"}}

            ]
}
```

2.13 A-EXPRESSION EVALUATION ALGORITHMS

As the examples above imply, algorithms are needed for evaluating an A-Expression to all or some row values within a field, or in some cases to tables. These algorithms implement operators over the registry, and therefore, we refer to them as *registry operators*. The algorithms outlined in this section are implemented and evaluated over the ontology stored in the knowledge registry (1) to resolve to fields or tables based on A-Expressions, and (2) to obtain all the dimension sets reachable through data operators from a given dimension set. These algorithms use the notations *ALL_TABLES*, *ALL_FIELDS*, *ALL_DIMENSIONS*, and *ALL_DIMENSIONSETS* to refer to all the tables, dimensions, and dimension sets in the knowledge registry. Note that A-Expressions use *ALL* to denote any of these.

These algorithms will be used by users (analysts, analytic tool developers, knowledge engineers) for multiple purposes. First, we describe below a list of the different types of such algorithms that need to be implemented to resolve A-Expressions based on fields and tables. The algorithms are described using Java-like syntax, and these algorithms implement registry operators in A-Expressions. Additional algorithms may be created for convenience, but these are the minimum registry operators or algorithms that are required to be implemented. Below, the bolded italicized phrases are the algorithm names. In parentheses, we describe the corresponding registry operator.

1. Given a list of tables, return a subset of those tables that match a given dimension set (/ registry operator)

```
public List<Table> matchTablesDimensionSetTables(List<Table> in_tables, DimensionSet ds) {
        List<Table> matchedTables = new ArrayList<Table>();
        // for each table check the all dimensions in dimensionset match
        // a subset of the fields in the table
        for (Table t : in_tables) {
                int matchesNeeded = ds.getDimensions().size();
                for (Dimension d : ds.getDimensions()) {
                        for (Field f : t.getFields()) {
                                if (f.getDimension() == d) {
                                        matchesNeeded--;
                                        break;
                                }
                        }
                }
                if (matchesNeeded < 1) {// we have a matching table
                        matchedTables.add(t);
                }
        }
        if (matchedTables.size() > 0)
                return matchedTables;
        else return null;
}
```

2. Given a list of tables, return a subset of fields that match a set of dimensions (* operation with dimensions as qualifiers)

```
public List<Field> matchTablesDimensionsFields(List<Table> l_tables,
                        List<String> l_dimensions) {
                List<Field> matchedFields = new ArrayList<Field>();
                for (Table t : l_tables) {
                        for (Field f : t.getFields()) {
                                for (String dimension : l_dimensions) {
                                        if (f.getDimension().getName().equals(dimension)) {
                                                matchedFields.add(f);
                                        }
                                }
                        }
                }
                if (matchedFields.size() == 0) return null;
                else return matchedFields;
        }
```

3. Given a list of fields, return a subset of fields that match a set of dimensions (* operation with dimensions as qualifiers)

```
public List<Field> matchFieldsDimensionsFields(List<Field> l_fields,
                        Set<Dimension> dimensions) {
                List<Field> matchedFields = new ArrayList<Field>();
                for (Field f : l_fields) {
                        for (Dimension dimension : dimensions) {
                                if (f.getDimension() == dimension)) {
                                        matchedFields.add(f);
                                }
                        }
                }
                if (matchedFields.size() == 0) return null;
                else
                        return matchedFields;
        }
```

4. Given a list of fields, return the fields that match a list of tags (* operation with tags as qualifiers)

```
public List<Field> matchFieldsTagsFields(List<Field> in_fields,
                List<Tag> in_tags) {
        if (in_tags.size() == 0)
                return null;
        List<Field> fields = new ArrayList<Field>();

        for (Field f : in_fields) {
                for (Tag tg : f.getTags()) {
                        for (Tag in_tag : in_tags) {
                                if (in_tags == null)
                                        continue;

                                if ((tg == in_tag)
                                 &  &  (tg.getTagScheme() == in_tag.getTagScheme())) {
                                        fields.add(f);
                                }
                        }
                }
        }
        if (fields.size() > 0)
                return fields;
        else
                return null;
}
```

5. Given a set of Fields A and a set of Fields B, return the fields that are in both sets (*intersectionFields*) (*&* operator)

6. Given a set of Tables A and set of Tables B, return the tables that are in both sets (*intersectionTables*) (*&* operator)

7. Given a set of Fields A and a set of Fields B, return the fields that are in either sets (*unionFields*) (| operator)

8. Given a set of Tables A and set of Tables B, return the tables that are in either sets (*unionTables*) (| operator)

9. Given a set of Fields A and a set of Fields B, return the fields that are in A but not in B (*minusFields*) (*!* operator)

10. Given a set of Tables A and set of Tables B, return the tables that are in A but not B (*minusTables*) (*!* operator)

As an example of mapping these algorithms to the operators, consider Example 4, *ALL * Dimension1 * (TagScheme1:Tag1 & (! TagScheme2:Tag1))* . We map *&, |,* and *!* to set operations intersection, union, and minus, respectively. The scope of *!* operator can be interpreted to be either as global, meaning all tags in all tag schemes, or just as all tags in the specified tag scheme, meaning tags in *TagScheme2*. We assume the global interpretation for A-Expressions with *!*.

The above A-Expression can easily be translated to the following prefix notation.

*(* (* ALL Dimension1) (& TagScheme1 : Tag1 (! TagScheme2 : Tag1)))*

By replacing the operators in the above expression with appropriate functions discussed earlier, we get:

(matchFieldsTagsFields (matchFieldsDimensionsFields ALL_FIELDS Dimension1) (intersectionFields TagScheme1 : Tag1 (minusFields ALL_FIELDS TagScheme2 : Tag1)))

2.14 QUALIFYING BY TIME PERIOD WITH . OPERATOR

One of the applications of the A-Expressions is to locate availability of data within a table. Sometimes data is available, and sometimes data is not. In such cases, registry maintains the availability information as shown in the *durations* field in the example below for tables. The fields in the table are available only within the durations described by start time and end times.

```
{
"tables":
      [
            {
      "name" : "EventTable1",
            "fields":  [ {"name": "f1"},{"name":"f2"} ],
            "durations": [ {"start_time": "2013-05-30T09:00:00", "end_time": "2013-05-30T10:00:00"}, {"start_time":
"2013-05-30T11:00:00", "end_time": "2013-06-01T10:00:00"}, {"start_time": "2013-07-02T10:00:00", "end_time":
"2013-07-05T10:00:00"}]},
   {
   "name" : "EventTable2",
    "fields" : [ {"name": "f3"},{"name":"f4"},{"name":"f5"}],
    "durations": [ {"start_time": "2013-05-30T09:30:00", "end_time": "2013-05-30T10:30:00"}, {"start_time": "2013-05-
30T11:30:00", "end_time": "2013-06-01T10:30:00"}, {"start_time": "2013-07-02T10:30:00", "end_time": "2013-07-
05T10:30:00"}]

   }
    ]
}
```

An additional registry operator, and an algorithm for it can be used to select a subset of fields that is available within a duration. This operation is *matchFieldsPeriodsFields*, and its algorithm is described below.

```
   public List<Field> matchFieldsPeriodsFields(List<Field> in_fields, List<Period> in_durations) {
            List<Field> fields = new ArrayList<Field>();
            for (Field f : in_fields) {
                  for (Period du: in_durations) {
                        if (f.getTable().fieldAvailable(f, du))
                              fields.add(f);
break;
                  }
            }
            if (fields.size() > 0)
                  return fields;
            else return null;
   }
```

This registry operator can be combined with other registry operators. An example based on Figure 8 is *ALL.{2013-05-30T09:12:00, 2013-05-30T09:33:00} * Dimension1*, and will return *EventTable1.Field1* and *EventTable1.Field2* using the tables described above.

Note that the duration information can also be stored for each field to describe unavailability of some fields of a table during some time periods, and similar algorithm as above can be applied to the fields of a table as well.

2.15 REACHABILITY IN A-EXPRESSIONS

It is possible and useful to specify all the dimension sets or sets of dimensions that can be reached through the data (store) operations. For example, COUNT() is a data operator. We define and implement two registry operators for analyzing reachability through the data operators. To process this reachability, the operator map table stored in the registry is used. We describe these registry operators with the example in Figure 8 in the previous section.

11. Given a dimension set, return the set of dimension sets that can be reached through data operations

```
public Set<DimensionSet> reachDimensionSetDimensionSets(DimensionSet in_ds) {
        List<Operator> startOps = new ArrayList<Operator>();
        for (Operator op : operators.getOperators()) {
                for (DimensionSet ds : op.getDimensionSetIn()) {
                        if (ds.getName().equals(in_ds.getName())) {
                                if (!startOps.contains(op)) startOps.add(op);
                        }
                }
        }
        Set<DimensionSet> dimSetsOut = new HashSet<DimensionSet>();
        List<Operator> allVisitedOps = getReachedOperators(startOps);

        for (Operator op : allVisitedOps) {
                dimSetsOut.add(op.getDimensionSetOut());
        }

        if (dimSetsOut.size() == 0) return null;
        else return dimSetsOut;
}
```

A registry operator corresponding to the above algorithm can be defined as *?* and can be used in A-Expressions. For example, to find all tables that can be reached through data operations from a dimension set, *DimensionSet11,* we could specify: *ALL/(DimensionSet11?),* which would be mapped to the prefix notation of *(/ ALL (? DimensionSet11))*, which in turn gets mapped to:

```
(matchTablesDimensionSetTables ALL (reachDimensionSetDimensionSets DimensionSet11))
```

A grammar can be defined for parsing A-Expressions, for example using a parser generator called ANTLR [7]. Appendix A has the ANTLR-based parsing rules. ANTLR can also be used to generate the registry operator to algorithm mapping described in the previous paragraphs.

2.16 USING A-EXPRESSIONS IN QUERIES

When an A-Expression may resolve a set of fields, or tables, the address expressing is effectively referring to all the data that is currently stored in those fields or tables. However, an analyst may be interested in the value in a single row or cell in the field. To specify subset of a field, or to apply additional processing on the data in one or more fields, A-Expressions may be used with data operators in queries. Below, we give an example of a Knowledge Query Language (KQL) query similar to what was described in [6], and show how the same query may be expressed in SQL with embedded A-Expressions.

3. KQL QUERY

An example in KQL query similar to the query in the previous report [6] is below.

```
{
   "OPERATOR" : "select",
   "INPUT" : [ {
            "DIMENSION" : "dest:domain",
            "VALUE" : "twitter.com"
   }, {
            "DIMENSION" : "Time",
            "VALUE" : "20131216060000,20131216065915"
   } ],
            "OUTPUT" : [ {
            "DIMENSION" : "fqdn"
   }, {
            "DIMENSION" : "ipv4"
   } ]
   } ],
            "EVENT" : "event:webwasher"
}
```

This query returns values of *fqdn* and *ipv4* dimensions from webwasher event with domain value is *twitter.com* that is a destination (*dest*) within the specified time period.

We can create an A-Expression by making a tag of *dest* from the default tag scheme, _ . The corresponding SQL query with A-Expression fragments will be:

```
SELECT {fqdn,ipv4}*_:dest FROM ALL/webwasher
WHERE domain.{ 20131216060000,20131216065915} ="twitter.com";
```

The input A-Expressions corresponding to the query is *ALL/webwasher* * *domain.* {*20131216060000, 20131216065915*} and the output A-Expression is *ALL/webwasher*{fqdn, ipv4}*_:dest*.

Note that a * operator is inserted in the input A-Expression to create the A-Expressions after the content in the FROM clause, and before the content of the WHERE clause. Similarly, a * operator is inserted after the content in the FROM clause, and before the content of the SELECT clause (*{fqdn ,ipv4}*_:dest*).

If there are multiple clauses in the WHERE clause joined by SQL logical operators (AND/OR/NOT), then there will be as many input A-Expressions as there are distinct A-Expression fragments in the WHERE clause. We only discuss embedding A-Expressions in the SELECT statement in SQL in this report, though same approach may be used to embed A-Expressions in other SQL statements.

The parse tree corresponding to the input A-Expression is below, created by defining grammar rules (see Appendix A) for parsing A-Expressions.

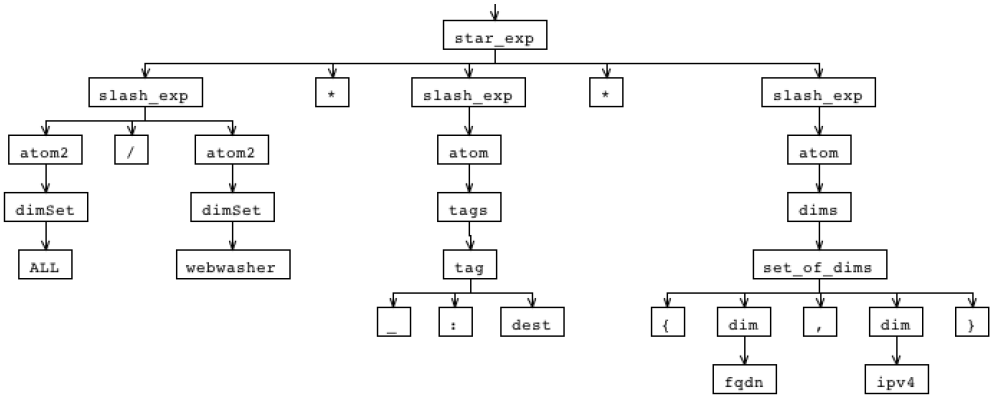

Figure 9. Parse tree for input A-Expression.

The parse tree corresponding to the output A-Expression is as follows:

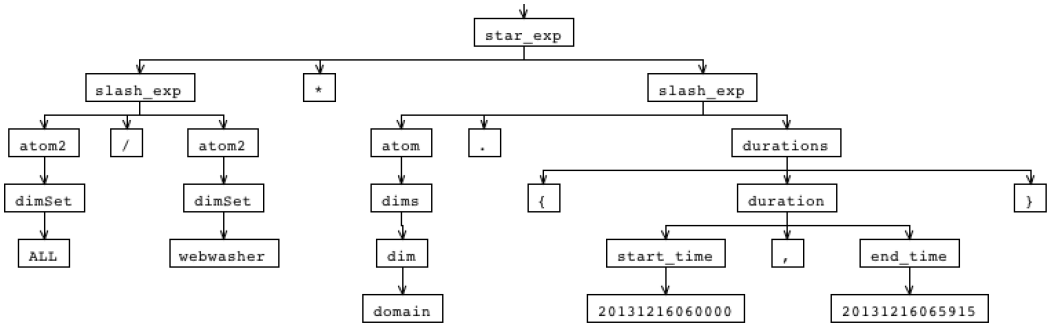

Figure 10. Parse tree for an output A-Expression.

Let us say the Table corresponding to this query is *Domain_tbl* and the dields for *fqdn*, *ipv4*, and domain dimensions are *fqdn_f*, *IPv4_f*, and *domain_f* respectively. Then, the corresponding SQL query that will be executed on the data store will be:

SELECT fqdn_f, ipv4_f FROM Domain_tbl
WHERE domain_f ="twitter.com" AND Start_time = 20131216060000 AND End_Time = 20131216065915;

We don't assume in this query that start and end times are also columns in *Domain_tbl*, though we chose to express the query that way. Different databases implement searching for time differently, and we expect the query execution engine to do the necessary conversion to the underlying executed query corresponding to the data store.

3.1 PROVENANCE RECORDING

Provenance recording is done within AQP (see Section 2) whenever an output A-Expression extracted from the query is processed. Since a query may have multiple output A-Expressions, the provenance record of the query result is the aggregation of the provenance records of all the individual output A-Expressions. Below, we describe provenance recording for each such output A-Expressions.

Provenance record for a dimension set, since a dimension set resolves to a table, comprises of the data collection objects corresponding to the table. For example, in Figure 11, the provenance record for A-Expression *ALL/DimensionSet15,* may have data collection sets <*DimensionSet15_DC1*, *DimensionSet15_DC2*>, which are two separate data collection objects corresponding to *DimensionSet15*. We assume *DimensionSet15* is not a derived dimension set from any other dimension set, and we consider the provenance record of a derived dimension set later in this section. Only a subset of the data collection objects of a table may be in provenance record if duration (.) registry operator is used in a A-Expression.

Since an A-Expression may evaluate to fields, and data collection entities (see Figure 5 and also [6]) in the registry ontology map to only tables, we will record provenance for a dimension to be the same as the provenance for the table in which the field occurs. Thus, provenance record for an A-Expression is the set of data collections corresponding to a set of tables will form the provenance record, if the tables and fields are not derived. For example, in Figure 11 (assuming *DimensionSet15* is not derived), the provenance record for A-Expression *ALL/DimensionSet15 * Dimension2* is <*DimensionSet15_DC1*, *DimensionSet15_DC2*>, which are two separate data collection objects corresponding to *DimensionSet15*. Provenance record for *DimensionSet15* contains <*DimensionSet15_DC1*, *DimensionSet15_DC2*>, the data collection objects for *DimensionSet15*. We assume *DimensionSet15* is not a derived dimension set.

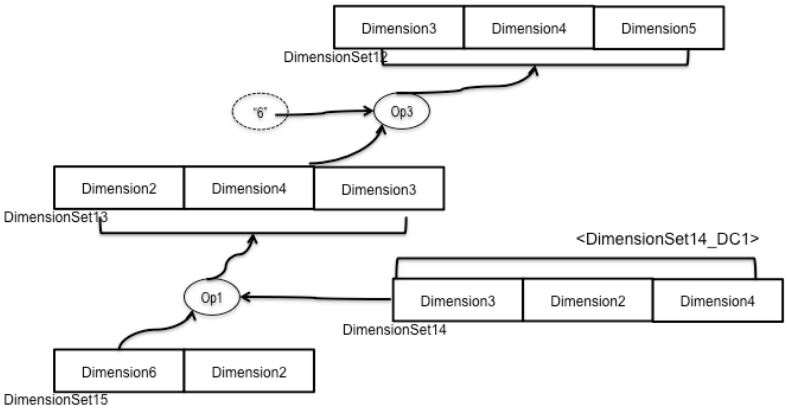

Figure 11. Provenance example.

If a dimension set is derived, then the provenance record for a dimension set (or for a dimension in the dimension set) will include one provenance path corresponding to each such non-derived dimension set from which that dimension set is derived.

Again, consider Figure 11. The provenance record for *ALL/Dimension12 * Dimension3* has two provenance paths.

{<<DimensionSet15_DC1,op1>, <DimensionSet15_DC2,op1>,
<DimensionSet13,op3>,<DimensionSet12>>,

<<DimensionSet14_DC,op1>,<DimensionSet13,op3>,<DimensionSet12>>}

Here *DimensionSet12* is a derived dimension set, and therefore appears as the last entry in the provenance path. The two non-derived dimension sets are *DimensionSet15* and *DimensionSet14*, and the data collection objects corresponding to those dimensions are the respective first entries in the corresponding provenance path. *DimensionSet13* appears in the path with *op3* because *DimensionSet13* is derived from *DimensionSet15* and *DimensionSet14* using *op3*.

Thus, a provenance record for a derived dimension set (or for one of its dimensions) consists of one or more provenance paths. Each provenance path is a sequence *<<DataCollection_1*, *data operator>*, …, *<DataCollection_n*, *data_operator>*, *<DimensionSet_X*, *data operator>*, …, *<DimensionSet_F>>*. The initial entries in the provenance path are the data collection objects corresponding to a non-derived dimensionset from which *DimensionSet_F* is derived. The next entries such as *DimensionSet_X* are the names of the dimension sets derived from the non-derived dimension set corresponding to those data collection objects. The names of these dimension sets will follow the sequence of derivation until

DimensionSet_F is derived. The final entry, *DimensionSet_F*, is the derived dimension set that A-Expression resolved to in this provenance path.

It is possible that some tables do not have a corresponding dimension set assigned, and a dimension set is derived from that table. In such cases, instead of the dimension set, the name of the table is recorded in a provenance path.

The provenance record is reported by AQP along with the query results.

This page intentionally left blank.

4. ADDITIONAL A-EXPRESSION EXAMPLES

In this section, we provide more A-Expression examples to help clarify the concepts already discussed so far.

4.1 DIMENSION AND * OPERATOR

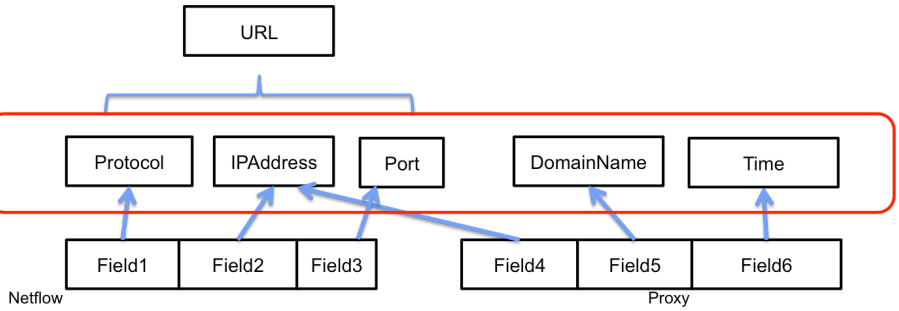

Figure 12. Netflow and proxy tables.

*ALL * IPAddress* resolves to the following fields: *Netflow:Field2* and *Proxy:Field4*.

*ALL * DomainName* resolves to the following field: *Proxy:Field5*.

4.2 DIMENSIONSET AND / OPERATOR

In Figure 12, how do you write an A-Expression that resolves to just *Netflow:Field2*? We need the concept of a dimension set to do that. Let there be two dimension sets,

DimensionSet1: *{IPAddress, Port, Protocol}*

DimensionSet2: *{IPAddress, DomainName, Time}*

Now,

*ALL/DimensionSet1 * IPAddress* resolves to only *Netflow:Field2*, and

*ALL/{IPAddress, Port, Protocol} * IPAddress* resolves to only *Netflow:Field2*.

Note that the two A-Expressions above are not the same. *{IPAddress, Port, Protocol}* is not *DimensionSet1* even though it has the same dimensions! It is a transient dimension set defined in the A-Expression. Note that resolving a dimension set to fields in a table requires every dimension in that dimension set to map to at least one field in the table. Also, the dimensions in a dimension set are not

required to map to any existing fields in existing tables implying that dimension sets have no obligation to match to any existing dimensions.

4.3 TAGS AND TAG SCHEMES

In Figure 12, how would you create an A-Expression to resolve just to *Proxy:Field4*? We need the concept of tags and tag schemes to accomplish this.

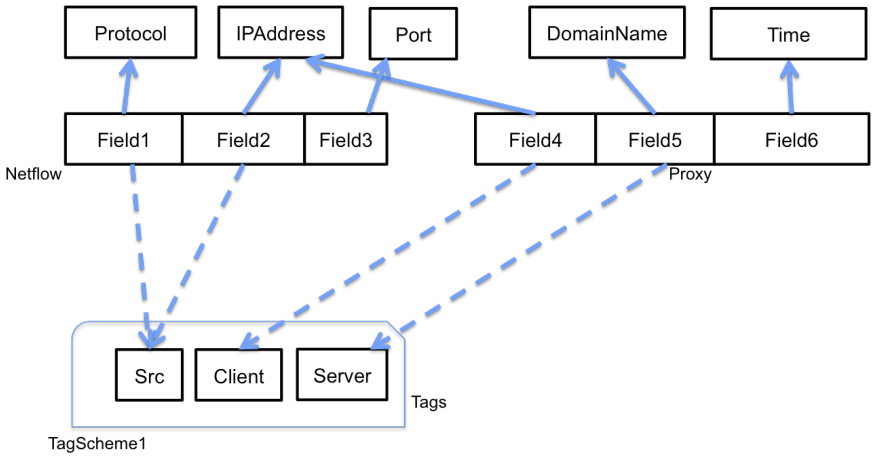

Figure 13. A single tag scheme.

Using tags in *TagScheme1*, the following A-Expression can be written to resolve just to *Proxy:Field4* as follows: *ALL * IPAddress * TagScheme1:Client*. The reason why this works is that *Proxy:Field4* is the only field that has the dimension of *IPAddress* and tag of *Client*. Can we create another A-Expression for *Proxy:Field4* using the schemes in Figure 13? It turns out that yes, we can. *ALL * IPAddress * (!TagScheme1:Src)* resolves to *Proxy:Field4* because the only field that has dimension *IPAddress* and does not have the tag *TagScheme1:Src* is *Proxy:Field4*. Figure 14 shows the addition of one more tag scheme, *TagScheme2*.

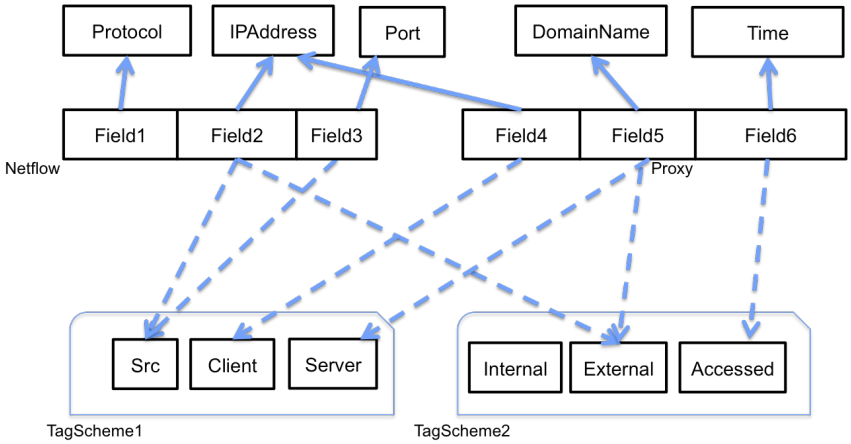

Figure 14. Two tag schemes.

Let's look at another example with tags. See Figure 15. What is an A-Expression for *EventTable2:Field4*?

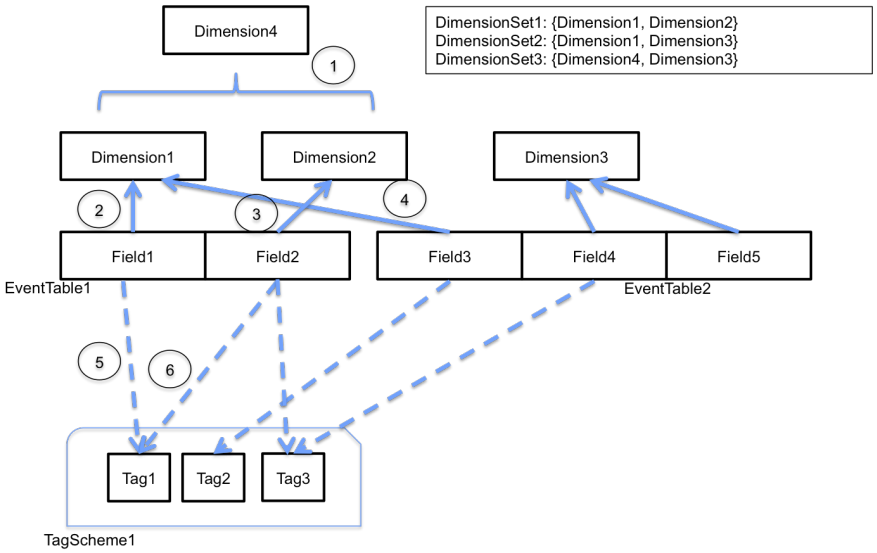

Figure 15. Another tag example with a single tag scheme.

*ALL * (TagScheme1:Tag3) * Dimension3* resolves to *EventTable2:Field4*.

Consider another example using Figure 16. Can an A-Expression be created for *Field1* without using a dimension set?

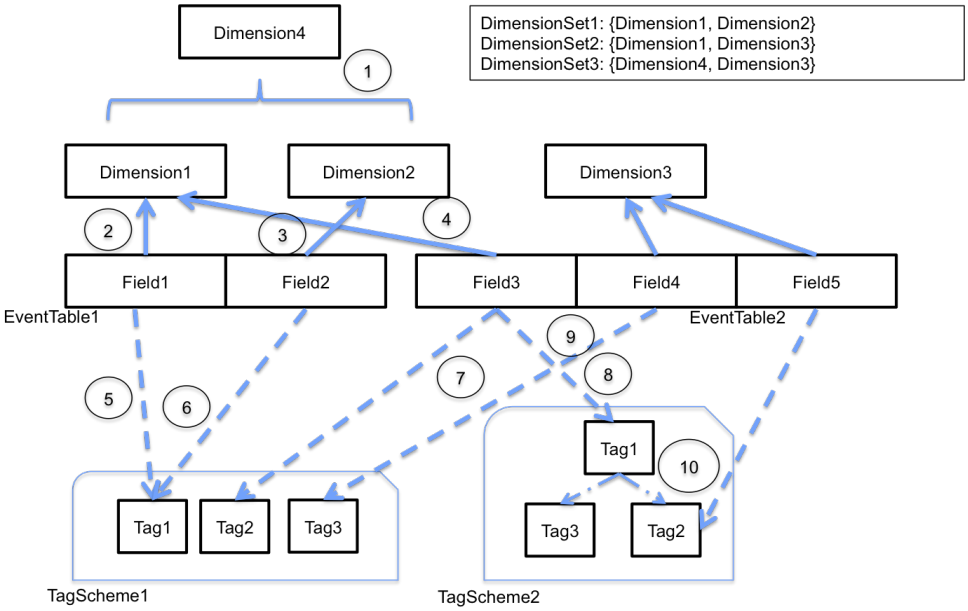

Figure 16. Another tag example with two tag schemes.

Yes. *ALL * (!TagScheme2:Tag1) * Dimension1* resolves to *EventTable1:Field1*.

4.4 REACHABILITY OPERATOR

Consider Figure 17. What does *DimensionSet15?* resolve to?

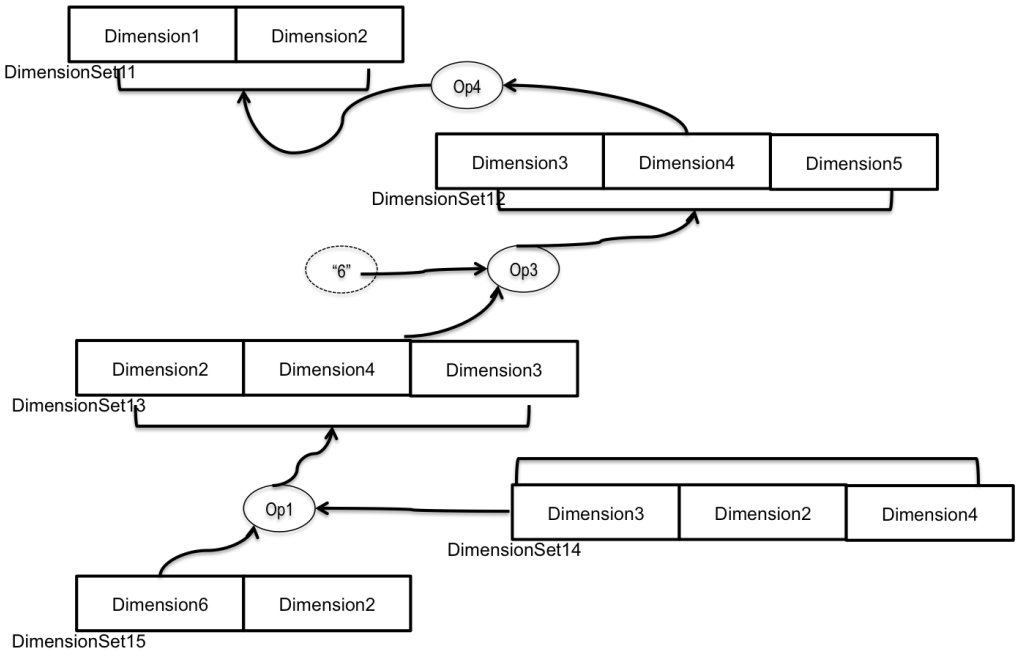

Figure 17. Reachability operator example.

It resolves to *DimensionSet13*, *DimensionSet12*, and *DimensionSet11*.

This page intentionally left blank.

5. CONCLUSION

This report describes techniques for accessing data stores using a knowledge registry by (1) adding tag-based, customizable, storage-independent addressing schemes for fields and tables in a key/value data store, and (2) defining and providing composable expressions called A-Expressions that can be integrated with existing query languages such as Structured Query Language (SQL) for addressing tables and fields. The concept of customizable tags and composable addressing schemes do not exist currently for data stores, either in key/value stores or traditional relational databases. The major benefits of this approach are listed below.

1. Ability to create and process complex A-Expressions using multiple registry operators. This capability allows creation of A-Expressions that include dimensions, dimension sets, tags, and tag schemes.

2. Ability to map semantic entities such as dimensions, dimension sets, tags, and tag schemes to tables and fields in the data store through A-Expressions. Users need not be aware of what tables exist, or where the tables physically exist in the data store. This feature means that a current query can be used at a later time to retrieve data from tables that do not yet exist.

3. Ability to provide provenance and context information such as which tables and fields are used for evaluating each A-Expression in a query, along with the results of a query.

4. Ability to embed and process the A-Expressions in popular and existing query languages such as SQL, thus making the queries declarative. The declarative and implementation-independent approach makes the queries more portable across different types of data stores and programming languages than a query embedded in a program snippet using a custom API.

5. Ability to customize the semantic mapping using tags and tagging schemes pertaining to domains such as network traffic, and further customizing the tag-based semantic mapping to specific user groups or even to individual users. These tags and tagging schemes may be stored in the knowledge registry, or elsewhere, so that A-Expression Query Processor (see Figure 3) may access it. In this report, we assume the tags and tag schemes are stored in the knowledge registry.

6. Ability to search the knowledge registry and discover the existence of tables derived from other existing tables using A-Expressions. The derivation information of tables may be stored in the knowledge registry, or elsewhere, so that A-Expression Query Processor (AQP) may access it.

7. Virtual dimension allows reinterpretation of existing ingested data without the need to redo data ingesting of data sources. Note that virtual dimension does not change the data store itself, just the knowledge registry. This is a huge advantage over traditional ways of attaching types to data

in columns either in relational databases or key/value stores. Virtual dimensions do not require discarding the old interpretation in order to use the newer interpretation.

8. Due to the separation of the data storage mechanism and the address resolution mechanism, it is possible to describe a new dimension set based on a data operation over other dimension set(s) dynamically, i.e., after the system goes into operation, without requiring to stop the ingesting platform.

9. The overall benefit of using our technique of ontology-assisted addressing of fields and tables is that ad hoc queries can be created by users with no knowledge of the fields or columns in the implemented data store tables, yet learn exactly which tables and fields were used to construct the results through associated provenance information.

APPENDIX A:
A-EXPRESSION PARSING RULES

```
grammar aexp17;

options {
output=AST;
ASTLabelType=CommonTree;
}

tokens {
ALL='ALL';
AND='&';
OR= '|';
NOT='!';
}

@lexer::header{
package edu.mit.ll.antlr;
}

@parser::header{
package edu.mit.ll.antlr;
}

@members{
boolean debug = false, caseInsensitive =false;
public void enableDebug(boolean value){
this.debug = value;
}
public void enableCaseInsensitive(boolean caseInsensitive) {
this.caseInsensitive = caseInsensitive;
}

}
start
  :
  r_exp
  {
  if(this.debug)
    System.out.println($r_exp.tree==null?"null":$r_exp.tree.toStringTree());
```

```
  }
  ;

r_exp
  :
  period_exp ((AND^|OR^) period_exp)*
  |
  NOT r_exp ->^(NOT r_exp)
  ;

period_exp
  :
  star_exp  ('#'^ star_exp)*
  ;

star_exp
  :
  slash_exp ('*'^ slash_exp)*
  ;
slash_exp
  :
  atom2 ('/'^ atom2)+
  |
  atom2 ('?'^)+
  |
  atom ('.'^ durations)*
  ;
atom2
  :
  set_of_dims ('.'^ durations)*
  |
  dimSet ('.'^ durations)*
  ;

atom
  :
  dims
  |
  tags
  |
  '(' r_exp* ')'
  {
  if(this.debug)
```

```
  System.out.println("Encountered rexp in parenthesis");
  }
  |
  '[' (dimSet|set_of_dimSets) ']'
  ;

dims
  :
  ALL
  |
  dim
  |
  set_of_dims
  ;

set_of_dims
  :
  '{' dim '}' -> '{' dim '}'
  |
  '{' dim (',' dim)+ '}'  -> '{'^(','dim+)'}'
  ;

set_of_dimSets
  :
  '{' dimSet '}' -> '{' dimSet '}'
  |
  '{' dimSet (',' dimSet)+ '}'  -> '{'^(',' dimSet+)'}'
  ;
dim
  :
  ID
  ;
tags
  :
  '{' tag '}' -> '{' tag '}'
  |
  '{' tag (',' tag)+ '}' -> '{'^(',' tag+)'}'
  |
  tag
  ;
tag
  :
  (tagScheme|'_') ':' ID
```

```
  ;

tagScheme
  :
  ID
  ;

dimSet
  :
  ALL
  |
  ID
  ;

durations
  :
  '{' duration (';' duration)* '}' -> '{'^(';'duration+)'}'
  ;

duration:
  start_time ',' end_time
  ;
start_time
  :
  INT -> INT
  ;
end_time
  :
  INT -> INT
  ;

INT : '0'..'9' ('0'..'9')*
  ;
ID  : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')*
    ;

COMMENT
    :   '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;}
    |   '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
    ;


WS  :   (' '|'\t')+ {skip();} ;
```

```
NEWLINE:'\r'? '\n' ;

STRING
    :  '"' ( ESC_SEQ | ~('\\'|'"') )* '"'
    ;

fragment
HEX_DIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;

fragment
ESC_SEQ
    :   '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\')
    |   UNICODE_ESC
    |   OCTAL_ESC
    ;

fragment
OCTAL_ESC
    :   '\\' ('0'..'3') ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7') ('0'..'7')
    |   '\\' ('0'..'7')
    ;

fragment
UNICODE_ESC
    :   '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
    ;
```

This page intentionally left blank.

BIBLIOGRAPHY

1 Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E., "Bigtable: A Distributed Storage System for Structured Data." In *Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation – Volume 7* (Seattle, WA: 6–8 November 2006). USENIX Association, Berkeley, CA, p. 15.

2 Doulkeridis, C., and Nørvåg, K., "A Survey of Large-Scale Analytical Query Processing in MapReduce," *The VLDB Journal*, pp. 1–26, 2013.

3 "Pig Latin Reference Manual 2," Hadoop, https://pig.apache.org/docs/r0.7.0/piglatin_ref2.html/.

4 "Apache Hive TM," Hadoop, https://hive.apache.org/.

5 Damodaran, S.K and O'Gwynn, D.B., "Method and Systems for Enhanced Ontology Assisted Querying of Data Stores," U.S. Patent Appl. No.:14/546,355, filed on 18 November 2014.

6 Damodaran, S.K., Yu, T., and O'Gwynn, D.B., "Knowledge Registry System and Methods," U.S. Patent Appl. 20150199424, issued on 16 July 2015.

7 Parr, T., "The Definitive ANTLR Reference: Building Domain-Specific Languages," *The Pragmatic Bookshelf*, 2007.

This page intentionally left blank.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
1 February 2016	Technical Report	

4. TITLE AND SUBTITLE

Knowledge Query Language (KQL)

5a. CONTRACT NUMBER
FA8721-05-C-0002 & FA8702-15-D-0001

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Suresh K. Damodaran

5d. PROJECT NUMBER
2231

5e. TASK NUMBER
61

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420-9108

8. PERFORMING ORGANIZATION REPORT NUMBER

TR-1199

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Bernadette Johnson, Chief Technology Officer
MIT Lincoln Laboratory
244 Wood Street
Lexington, MA 02420-9108

10. SPONSOR/MONITOR'S ACRONYM(S)
ASD R&E

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution Statement A: Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Currently, queries for retrieval from NoSQL datastores are tightly coupled to the specific implementation of the datastore implementation, making portability of the queries or query-dependent algorithms difficult. This report introduces a declarative approach that is independent of the storage content and format for querying NoSQL or relational data stores. This approach uses *address expressions* (or A-Expressions) embedded in commonly used query languages such as Structured Query Language (SQL). The declarative approach makes the queries portable, and results in several advantages over the existing approaches to querying, especially when the data is semi-structured, and when the data sources may change over time. Cyber event logs are examples of such data sources. When the query is independent of the underlying physical data sources, having provenance information on the query results becomes important to impart necessary context, and ensure trust in the query results returned. This declarative approach is made possible through the use of a Knowledge Registry. In this report, we discuss embedding A-Expressions in the widely used SQL, resolving A-Expressions using the ontology implemented in a Knowledge Registry, and returning query results with provenance information.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	Same as report	57	**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified			

This page intentionally left blank.