Data Query Language: A DSL for Vetted Queries

DQL Developers

ABSTRACT

One of the main goals of OPAL (Open Algorithms) project is to enable seamless data sharing between arbitrary entities, while at the same time ensuring that query results do not leak sensitive information. For instance, if an identity provider queries a data store, it should be able to obtain results that allow it to create assertions about an identity (such as *id A belongs to group G*) and nothing else. In particular, it should not be able to obtain personally identifying information about the individual. OPAL attempts to do this by restricting the type of allowed database queries. Additionally, OPAL aims to bring together heterogeneous data stores, each with its own query languages. There are several dialects even with SQL. Due to this diversity, defining a common framework for privacy preserving data sharing seems difficult because a query auditor will have to understand several syntax and semantics. To address this, OPAL defines its own query language called DQL (Data Query Language) which is data-store agnostic. Similarly to SQL, DQL can be used to represent arbitrary relationships between data, create more views of the data and finally query those views. DQL is expressive, yet compact and can capture SQL queries spanning several lines in a single statement. Another goal of DQL is to allow us to prove certain assertions about information leakage.

1. INTRODUCTION

The OPAL project aims to enable competing organizations to come together and share data in order to achieve some common goal. The health sector (for instance cancer research) seems to me one benefactor of such technology. Data sharing in clinical trials for cancer research is governed by strict privacy laws and consequently a large amount of time and money is spent on repetitions of trials generating similar data. If such data was already available, many of these tests could be skipped. Thus, any data-sharing in such scenarios has to go through extensive scrutiny for possible leakage. Not only does this entail technical skills but also legal expertise. OPAL aims to address this issue by requiring:

1. Queries should return only aggregate answers and never information about individual rows.
2. Only vetted queries should be allowed. A vetter would publish templates of allowed queries.
3. A query auditor analyzes queries created from the vetted templates to ensure no private information is leaked.
4. The entire transaction of query/response is be recorded in a blockchain for audit trails.

1.1 Query Auditing

The following definitions are taken from the literature. A *dataset* is a set X of labeled real numbers $\{x_i, x_2, \ldots x_n\}$ where n is public. A query q consists of a pair (f, s), where $f \in \{\mathsf{max}, \mathsf{min}, \mathsf{avg}, \mathsf{sum}\}$ representing the aggregate functions and $s \in [1..n]$. The response to q is a real number representing the aggregate applied on the subset of X defined by s. An *offline auditor* takes in a sequence of query-responses and decides if the responses leak any sensitive data. An *online auditor* takes in a sequence of query-responses and additionally a query q along with its response r to decide if r might leak sensitive data (it does not care if the past responses leaked information). If an online auditor decides that r leaks information then r can be denied. However, as shown in [**?**], the very act of denying a response can leak information. A *simulable auditor* is an online auditor that takes this decision without looking at r. Hence, an adversary can also simulate the auditor's role and we can be ensured that denials do not leak information.

The concept of information leakage is either captured via *full disclosure*, where an adversary is able to compute some or all of x_is or *partial disclosure*, where the adversary is able to narrow down on the range of some (or all) x_is. Note that the full disclosure definition is too weak to be useful in practice because even if the adversary cannot compute the exact value, a narrow enough range may be sufficient for leakage. On the other hand, partial disclosure is too strong to be achievable in practice, because every useful query must leak *some* range information. Note that we do not consider the trivial case of denying all queries since that indeed achieves resistance against partial disclosures.

1.2 Query Vetting in OPAL

OPAL uses slightly modified definitions. An OPAL dataset is a set X of labeled real numbers $\{x_i, x_2, \ldots x_n\}$ where n is **secret**. An OPAL query q consists of a pair (g, t), where

$g \in \{\mathsf{max}, \mathsf{min}, \mathsf{avg}, \mathsf{sum}, \mathsf{count}\}$ representing the aggregate functions and $t \in A$ for some $A \subsetneq 2^{[1..n]}$, the power set of $[1..n]$. We call A the *set of allowed queries* of X.

An *OPAL query template* is a mapping from the data set X to an allowed query set A. An *OPAL vetter* is responsible for defining the query template, while an *OPAL auditor* decides if a given query q is indeed an element of A.

The OPAL definition are more usable because typical queries such as `select max(age) from users where state = 'MA'` do not allow the adversary to select arbitrary subsets of X but rather those defined by a filter (`state = 'MA'`). Thus, one approach to query vetting would be to restrict the filters that can be applied. For instance, a filter to ensure that n remains secret would need to ensure that the adversary can never select all or exclude a fixed number of elements from a query set.

1.3 OPAL Query Language

In order to define such a query framework and filter rules for allowed queries, we use a query language called DQL, which is specifically designed for working with sets and filters. The prototype implementation of DQL uses Datalog as its underlying query engine but adapters for converting to standard SQL can be created. The key point is that once a DQL query is considered safe by a vetter or auditor, we don't have to worry about vetting the corresponding SQL or another low-level query. Hence we can focus our security analysis on DQL while perform the actual query in another language if needed.

2. OVERVIEW OF DQL

DQL is a language for querying structured data stored in a set of *tables*. A table in DQL is called a *pattern*. A pattern has a name, a *schema* describing the data and zero or more rows containing the data. Pattern names start with #, such as `#users`. DQL queries are of two types: a `def` query defines new patterns (similar to creating views in SQL) and a `find` query returns row IDs of a pattern (similar to a select query in SQL).

Schema: A data store makes its data available by first publishing the schema in PDL (Pattern Definition Language), an example of which is:

users(name:String, userID:String[uID], stateID:String[sID], age:Int)
state(name:String, id:String[sID], capital:String)

The above defines two patterns `#users` and `#state` with various attributes. Additionally, the `stateID` attribute of `#users` and the `id` attribute of `#state` can be used in a join because they map to the common key `[sID]`. This published schema is called the *basis*.

Comparing with SQL: Since DQL is strictly a query language, it does not provide methods to insert or update rows. However, note that a DQL `def` query is equivalent to creating a view in SQL. Similar to SQL, DQL is a *set algebra*, a language that operates on sets and provides the ability to perform operations on arbitrary subsets of those sets. In SQL however, the programmers needs to specify *how* to do the operations by specifying the joins. In DQL, the joins are automatically inferred, allowing the programmer to focus on *what* rather than *how*. As an example, the DQL:

```
find #users where {#state.@capital = 'Springfield'}
```

can map to the SQL:

```
select * from users where
          user.stateID = state.stateID and
          state.capital = 'Springfield'.
```

The joins are automatically inferred from the DQL schema.

DQL is expressive enough to represent all necessary relationships found in most SQL databases and additionally allows recursive rules to be defined

3. THE DQL FRAMEWORK

The DQL framework comprises two primary languages:

1. *Query language:* This is the primary component, used for querying data with some user-specified criteria. This criteria is specified in what we call the DQL syntax. Its code is given in `typeface` font, as in: **`def #human as #person`**.

2. *Schema language:* The initial schema is specified via **PDL** (Pattern Definition Language). PDL code is given in *italics*, as in: *person(name:String, age:Int)*.

Terminology: We use the following terminology:

1. **Tables:** DQL operates on primitives called *tables*. Table names always start with '#' (example `#person`). Users define new tables by extending existing ones.

2. **Basis:** These are initial tables defined via PDL.

3. **Attributes:** A basis table has a name and one or more attributes. An attribute is a typed variable and starts with '@'. As an example, the PDL code:

   person(name:String, age:Int, city:String)

   defines a basis table `#person` with three attributes: `@name`, `@city` (`Strings`), and `@age` (`Int`). User-defined tables do not have attributes.

Figure 1 summarizes the various components.

Language	*Purpose*	*User*	*Ref.*
DQL	Define new tables	Data seeker	3.1
PDL	Define initial tables	Data owner	3.2

Figure 1: DQL framework components

3.1 DQL Syntax

DQL is the main language of the framework and allows users to do two things: (1) define new patterns by extending one or more existing patterns and (2) find rows in patterns. Here we explain both. See Appendix E for the DQL grammar.

Defining Patterns: A new pattern is defined as

`def <newPattern> as <match>`

Here `<match>` is a combination of one or more patterns and optional filters. We illustrate this via examples below. To improve readability in this section, DQL code will be in `gray` and the `<match>` expression will be in **`bold`**. The patterns inside `<match>` are called the parents of `<newPattern>`.

1. **Filter:** A filter extracts a subset of a pattern. For instance: `def #adult as #person where {@age > 18}`

   A filter is a tuple (*attribute*, *operator*, *value*) (in this case (`@age`, `>`, `18`)), specified using `where` keyword. The `~` and `~~` operators represent wildcard and regex respectively, for use with `Strings`.

2. **Merging filters:** Multiple filters can be used, as in: `def #member as #person where {@age > 25 or (@city = 'Springfield' and @age > 20)}`

   Filters are merged via **and**/**or** using parenthesis.

3. **Traversing patterns:** User-defined patterns do not have attributes. However, attributes of any basis pattern can be accessed using `.` operator:

   `def #elder as #member where {#person.@age > 30}`

   This is called traversal – we 'traverse' from `#member` to `#person` and access `@age`.

4. **Patterns as values:** Some attributes can map to a pattern. Example, `@child` takes value `#adult` in:

   `def #senior as #parent where {@child = #adult}`

   The rules for this are described in section 3.2.2.

5. **Attributes as values:** Any attribute can map to another attribute of the same type, as in:

   `def #MD as #person where {@name = #corp.@founder}`

6. **Merging patterns:** `<match>` can have multiple patterns merged via **and**, **or**, **not** or **xor** (denoting respectively, intersection, union, set difference and symmetric difference operations), as in: `def #eligible as {#person where {@age>20} and {#member or #MD}}`

   Merged patterns must be enclosed in braces.

Finding Patterns: Results are returned using a `find` statement, the syntax of which is: **`find <match>`**, where `<match>` is any single-pattern expression. For instance: `find #eligible where {#person.@age < 50}`.

The DQL compiler takes in a sequence of `def` and `find` statements along with schema definitions and code such as SQL or Datalog for some underlying database.

3.2 Schema Definition

Recall that some initial patterns are mapped to the basis via configuration defined using PDL (Pattern Definition Language). Consider the earlier PDL example to define a basis pattern called `#person`:

person(name:String, age:Int, city:String) // PDL code

In reality, the above will give an error because in addition to attributes, each pattern must have either one or more *primary keys* or one or more *pattern keys*.

3.2.1 Primary Keys

Two patterns are linked via primary keys. Any attribute can be set as a primary key by appending an ID enclosed in square brackets to the attribute, as in the following PDL:

person(persID:String[pID], name:String, age:Int, city:String)

The above defines a primary key with ID `pID`, written as `[pID]`. This ID is used to match primary keys of other patterns. Now let us define two more patterns via PDL:

corp(corpID:String[cID], name:String, founder:String)
works(persID:String[pID], firmID:String[cID], empID:String)

Here `#corp` has one primary key `[cID]`, while `#works` has two primary keys, `[cID]` and `[pID]`. The two patterns are connected via the common primary key `[cID]`, similar to the way RDBMS tables are connected. However, there are no foreign keys in DQL and all such relationships are *many-to-many*. Primary keys have the following properties:

1. Primary keys attributes cannot be used in a filter. They are used internally for traversal and merging.
2. A pattern returns its primary keys (or pattern keys - see below). Therefore, `#person` will return `[pID]`s.
3. A new pattern inherits the (primary or pattern) keys of its parent pattern(s). Therefore, any pattern defined using `#person` will also return `[pID]`s.

Traversal: Traversal occurs when a pattern is extended and the filter accesses an attribute of a different pattern. The patterns must be connected by a chain of primary keys for traversal to be successful (otherwise an error will be thrown). If such a path exists then DQL will automatically join the relations (i.e., pattens) in the path using the primary keys. As an example, the following pattern, `#fromAcme` uses `#corp` directly and `#works` indirectly for traversal:
`def #fromAcme as #person where {#corp.@name='Acme'}`.

Traversal from `#person` to `#corp` is done via `#works` as follows: `#person[pID] -> #works[pID,cID] -> #corp[cID]`. The traversal is explicitly seen in the compiled Datalog code:

```
fromAcme(pID) :- person(pID, _, _, _), works(pID, cID, _),
                 corp(cID, 'Acme', _).
```

3.2.2 Pattern keys

Pattern keys define attributes that map to another pattern (see Example 4, Section 3.1). Pattern keys are defined by an ID enclosed in braces. The following defines a pattern with two pattern keys both with ID `pID`:

parent(person:String{pID}, child:String{pID})

A pattern key has the following features:

1. A pattern key attribute can be used in a filter. It can map to any pattern that returns a primary key with the same ID. In our example, both `@child` and

`@person` attributes of `#parent` can map to any pattern that returns `[pID]` (such as `#fromAcme`). Therefore, we can, for example, write:

```
def #AcmeParent as #parent where {@child = #fromAcme}
```

2. In a filter, a pattern returning a pattern key can be used in place of a pattern returning the same primary key. Therefore, in the code above the `@child` attribute can also take as input any pattern returning `{pID}`.

3. Pattern keys are non-traversable and cannot be merged with primary keys. Thus, a pattern returning `[pID]` cannot be merged with one returning `{pID}`.

Hiding returned pattern keys. A pattern returns all its pattern keys. For instance, `#parent` as defined above will return the pattern keys `{pID, pID}`. Suppose, we want it to return only the first key (representing parents), we can skip the other key(s) by appending '!' to it, as in:

parent(person:String{pID}, child:String{pID!})

After this, `#parent` will return only the first key `{pID}`, and the following code, for example, will be valid DQL:

```
def #grandParent as #parent where {@child = #parent}
```

Once the pattern schema, the primary and pattern keys are defined in PDL, the schema definition is complete. The set of initial patterns defined in the schema is called the *basis*. Advanced users can make additional tweaks to the basis configuration as described in Appendix C.

A DQL server will be initialized with a basis along with raw data for its patterns (supplied in CSV format). After this, it can accept DQL queries and return data based on the result mapping supplied. The DQL and the basis must adhere to semantic restrictions given in Section D.

Since DQL is primarily a query language, it does not have queries to insert data and all external data must be provided in the initialize phase. However, every `def` query in DQL effectively creates a temporary pattern internally.

3.3 Result Mapping

A DQL `find` query returns the primary or pattern keys of the pattern. These can be considered as unique row IDs. The query `find #users where {#state.@capital = 'Boston'}` could, for example, return following keys of type `[userID]`:

```
user01
```

These are not meaningful to end-users. The above row IDs can be mapped to meaningful values using a *Result Mapping*. As an example, the following lines define various result mappings from `userID`:

```
map :a as $userID => #users // all public columns
map :b as $userID => #users.@name // name
map :c as $userID => #users.@age.avg // avg of age
```

We can then specify a mapping in the = query as follows: `find #users:a where {#state.@capital = 'Boston'}`. The query would then return the following:

```
name |age
-----|---
Homer|45
```

The grammar for mapping is given in Appendix **??**.

4. RESTRICTING QUERIES

Constraint Language: In OPAL, we also need a way to restrict the type of queries to prevent data leakage. This is done by constraining the kind of DQL queries allowed. We can, for example, restrict certain columns of certain patterns (e.g., *age* in pattern *users*) and the operations on those columns (cannot do $<, >, \geq, \leq, \neq$). Additionally, the result mapping can be used to restrict the final view of the query results.

When specifying constraints, instead of following the standard blacklist approach (*deny these and allow rest*), we follow a whitelist one (*allow these and deny rest*). Thus, the constraints define the allowed (rather than denied) queries, and are usually created by the data-owner. If no constraints are specified, then all queries are disallowed. As an example, the following allows DQL queries that access the column *age* of pattern users with the operations $=, \neq$:

```
ageConstraint:  #users.@age:  =, !=
```

When making a query, the following needs to be specified:

1. A set Q of DQL queries of `def` and `find` statements.

2. A set C of whitelist constraints that accept the queries.

3. An set R of result mappings, one for each `find` query.

5. CONCLUSION

We described Data Query Language (DQL), a DSL that automates the generation and maintenance of generic database queries in a human-friendly syntax. [...]

6. REFERENCES

[1] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: An interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 1213–1216, New York, NY, USA, 2011. ACM.

[2] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1:146–166, 1989.

[3] Raghu Ramakrishnan and Jeffrey D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23:125–149, 1993.

[4] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

APPENDIX

A. OVERVIEW OF DATALOG

Datalog is a declarative programming language used for querying deductive databases. It has found applications in information extraction, program analysis and security in recent years [1, 2, 3]. In Datalog, table schemas – for example, `Parent(A, B)` – are called as *Relations* and the corresponding table rows – for example, `Parent('Homer', 'Bart')` – are

called *Facts*. New tables are constructed via *Rules*. Rules are made of two parts separated by the :- symbol: (1) One *Head relation* on the left, and (2) One or more *Tail relations* on the right each separated by the , symbol. For example:

```
GrandParent(A, C) :- Parent(A, B), Parent(B, C)
```

Datalog supports recursive rules such as:

```
Ancestor(A, C) :- Ancestor(A, B), Ancestor(B, C)
```

Tail relations in the same rule are combined using conjunction. Two or more rules with the same head relation – for example below – are combined using disjunction:

```
Ancestor(A, B) :- Parent(A, B)
Ancestor(A, C) :- Ancestor(A, B), Ancestor(B, C)
```

Comparison with Relational Algebra. Although Datalog has similarities to SQL/relational algebra, they are different. Any expression in basic relational algebra can be expressed in Datalog. Operations in extended relational algebra (grouping, aggregation, sorting) are not supported in Datalog. Datalog can express recursion, which SQL cannot.

B. DQL GRAMMAR IN ANTLR

The DQL compiler is written in Scala using ANTLR v3 [4], a tool for generating parsers. The following is the DQL grammar, created using ANTLRWorks (www.antlr3.org/works).

```
//  Lexer rules start with uppercase letter
//  Parser rules start with lowercase letter.
//
// .. creates a character range
// + means "repeat one or more times".
// * means "repeat zero or more times".
// ? means "optional" (i.e., "repeat zero or one time")
// | means "or"

tables:       'tables' ID ':' (define|find)* EOF;
find:         'find' simpleMatch;
define:       'def' TABLE 'as' tableMatch;
tableMatch:   simpleMatch | ('{' complexMatch '}');
simpleMatch:  TABLE filter?;
complexMatch: tableMatch boolMatch tableMatch;
filter:       'where' '{' rule '}';
rule:         simpleRule | complexRule;
mixedRule:    simpleRule | ('(' complexRule ')');
simpleRule:   attValStr | attValInt | attValTab | attValAtt;
complexRule:  mixedRule bool mixedRule;
boolMatch:    bool | 'not' | 'xor';
bool:         'and' | 'or';
attValStr:    ATT opStr STRING;
attValInt:    ATT opInt INT;
attValAtt:    ATT opEq ATT;
attValTab:    ATT opEq TABLE;
opEq:         '=' | '!=';
opInt:        opEq | '<=' | '>=' | '<' | '>' ;
opStr:        opEq | '~' | '~~';
ATT:          '@' ID | (TABLE '.' '@' ID);
TABLE:        '#' ID;
//ID,STRING,INT are defaults defined with AntlrWorks
//ID: anything starting with non-number, INT: number
//STRING: Anything enclosed in single quotes
```

C. DATALOG CONFIGURATION

A DQL database uses Datalog as its underlying query engine. Thus, advanced users familiar with Datalog can make additional use of its powerful features (such as recursion). Recall that the basis is the set of initial table schema. Here we describe how the basis can be enhanced with Datalog. First we split the basis into a *reduced basis* and an *extended basis*. The reduced basis is essentially the schema of tables mapping to the externally supplied raw data. The extended basis is derived from the reduced basis via Datalog rules and does not correspond to externally supplied data. We explain this by extending the earlier examples.

C.1 Reduced basis

Let us enumerate the basis tables for the examples of Sections 3.1 and 3.2:

person(persID:String[pID], name:String, age:Int, city:String)
works(persID:String[pID], firmID:String[cID], empID:String)
corp(corpID:String[cID], name:String, founder:String)
parent(person:String{pID}, child:String{pID})

The data for these tables is supplied by the user. All such tables form the *reduced basis*.

C.2 Extended Basis

Suppose we need to define tables for siblings and ancestors. Let us add them to the basis via PDL:

sibling(left:String{pID}, right:String{pID})
ancestor(ancestor:String{pID}, descendant:String{pID})

Their data can be inferred from the reduced basis and should not be user-supplied. These form the *extended basis*.

C.3 Initial Rules

Since the extended basis does not contain data, using it in DQL will return no results. Therefore, some rules need to be supplied to populate it using the reduced basis. These are the *initial rules*.

In the case of Datalog, our initial rules would be:

```
sibling(x,y) :- parent(p,x), parent(p,y), x != y.
ancestor(x,y) :- parent(x,y).
ancestor(x,y) :- ancestor(x,a), ancestor(a,y).
```

In the case of SQL, our initial rules would map to views:

```
sibling(x,y) :- parent(p,x), parent(p,y), x != y.
ancestor(x,y) :- parent(x,y).
ancestor(x,y) :- ancestor(x,a), ancestor(a,y).
```

Once the reduced basis, the extended basis and the initial rules are defined, the configuration is complete. The configuration is validated as described in Appendix D.

D. SEMANTIC RESTRICTIONS

The following rules and restrictions must be followed with respect to the basis and DQL code.

1. A basis table has at least one primary/secondary key and returns all its primary/secondary keys, with the exceptions described in Section 3.2.2.
2. A basis table cannot have both primary and secondary keys. Thus, a basis table with secondary key(s) is non-traversable. Using the definitions of Section C.1, the following code will give an error:

   ```
   def #foo as #parent where {#person.@name='Homer'}
   ```

   This is because #parent is non-traversable.

3. There cannot be multiple traversal paths in the basis. The following PDL code will give an error:
   person(persID:String[pID], name:String, age:Int, city:String)
   works(persID:String[pID],corpID:String[cID],empID:String)
   corp(corpID:String[cID], name:String, founderID:String[pID])
   This is because there are two paths from `[pID]` to `[cID]`; one via `#works` and another via `#corp`. Such relations can be modeled using secondary keys. As an example, the following is valid PDL code:
   person(persID:String[pID], name:String, age:Int, city:String)
   works(persID:String{pID},corpID:String{cID},empID:String)
   corp(corpID:String[cID], name:String, founderID:String[pID])
4. Two tables can be merged (using `and`/`or`/`not`/`xor`) if only if their returned keys are identical. Using the basis defined in Section C.1, the following will, therefore, give an error: `def #foo as {#person and #corp}`,
   as `#person` returns `[pID]`, while `#corp` returns `[cID]`.
5. A user-defined table returns the key(s) of its parent table(s). Therefore, a user-defined table is traversable if and only if its parents are traversable. As an example, `#fromAcme` of Section 3.2.1 is traversable and we can write:
   ```
   def #emp1 as #fromAcme where {#works.@empID='1'}
   ```
6. Type restrictions are enforced in DQL. Therefore, for example, using the definition of Section 3.2.1, the following will given an error because an attribute of type `String` cannot be mapped to an `Int`.
   ```
   def #emp1 as #fromAcme where {#works.@empID = 1}
   ```

Selecting The Basis. First set the reduced basis to represent the initial data. Then set the extended basis for other relations that may be needed. If there are loops in traversal, make some tables non-traversable by using secondary keys. Finally, make those tables non-traversable whose attributes need to map to other tables.

A basis table can be hidden from the user using '!' as in:
works!(persID:String[pID],corpID:String[cID], empID:String).

Such hidden tables cannot be used in DQL. However, they might be used internally for traversal.

E. MAPPING GRAMMAR IN ANTLR

The following is the mapping grammar in ANTLR3.

```
mappings:    'mappings' ID  (mapping)+ EOF;
mapping:     'map' MAP 'as' keyMaps ';';
MAP:         ':' ID;
keyMaps:     keyMap | (keyMap ',' keyMaps);
keyMap:      KEY^ '=>' value;
value:       COUNT | pattern;
pattern:     PATTERN ('.' countOrAttr (',' countOrAttr)*)?;
countOrAttr: COUNT | attribute;
attribute:   ATTR^ ('.' countOrAggr)?;
countOrAggr: COUNT | AGGR;
PATTERN:     '#' ID;
KEY:         '$' ID;
ATTR:        '@' ID;
AGGR:        'max'|'min'|'sum'|'avg';
COUNT:       'count';
```