Metadata Management for Scientific Databases

Pietro Pinoli[a], Stefano Ceri[a], Davide Martinenghi[a], Luca Nanni[a]

[a]*Politecnico di Milano*

Abstract

Most scientific databases consist of datasets (or sources) which in turn include samples (or files) with an identical structure (or schema). In many cases, samples are associated with rich metadata, describing the process that leads to building them (e.g.: the experimental conditions used during sample generation). Metadata are typically used in scientific computations just for the initial data selection; at most, metadata about query results is recovered after executing the query, and associated with its results by post-processing. In this way, a large body of information that could be relevant for interpreting query results goes unused during query processing.

In this paper, we present ScQL, a new algebraic relational language, whose operations apply to objects consisting of data-metadata pairs, by preserving such one-to-one correspondence throughout the computation. We formally define each operation and we describe an optimization, called *meta-first*, that may significantly reduce the query processing overhead by anticipating the use of metadata for selectively loading into the execution environment only those input samples that contribute to the result samples.

In ScQL, metadata have the same relevance as data, and contribute to building query results; in this way, the resulting samples are systematically associated with metadata about either the specific input samples involved or about query processing, thereby yielding a new form of *metadata provenance*. We present many examples of use of ScQL, relative to several application domains, and we demonstrate the effectiveness of the meta-first optimization.

Keywords: Metadata management, scientific databases, query optimization

1. Introduction and Motivation

The organizations of scientific databases are very different. In many scientific fields, such as biology and astronomy, big consortia produce large, well-organized data repositories for public use. In other contexts, such as public administrations, data are open but much less organized and much more dispersed. Other big data players, such as Internet companies or mobile phone operators, produce information mostly for internal use, but often support third parties in research studies (e.g., about consumers' interests) by providing them with services for data retrieval.

We abstract a scientific data source as a container of several datasets, that in turn consists of thousands of samples, one for each experimental condition, often stored as files and not within a database; typically, samples are described by *metadata*, i.e., descriptive information about the content and production process of each sample. In meteorology, typical metadata describe "the WDM station, the sources of meteorological data, and the period of record for which the data is available"; then the samples describe millions of records registered at the station. In genomics, typical metadata describe "the technology used for DNA sequencing, the process of DNA preparation, the genotype and phenotype of the donor"; then, samples describe millions of genomic regions collected during the experiment.

Metadata support the selection of the relevant experimental data by means of user interfaces (e.g. see genomic repositories such as ENCODE (the Encyclopedia of Genomic Elements, [1]) or TCGA (The Cancer Genome Atlas, [2]). When a source exposes APIs or WEB interfaces, metadata associated to each sample (such as Twitter's hashtags or timestamps) support data retrieval. As a result, data scientists select the best datasets and samples within sources after analyzing the metadata, but typically they do not further use them; in most cases, then they retrieve samples and use scripting languages to query and analyze them. In this paper, we present a new query language for scientific databases where metadata and samples are processed together, thereby allowing for important optimizations and also producing metadata describing query results.

Our approach is influenced by our experience in biological databases. In that domain, we developed GMQL (GenoMetric Query Language) [3], a language for the integration of heterogeneous biological datasets. Some of the lessons learned in that domain include aspects that can be shared and generalized to virtually all scientific databases, namely:

- Metadata describing the content and production process of a sample can be explicitly associated with the sample in the form of attribute-value pairs.
- Samples can be represented as parts of larger datasets (e.g., files in a file system or tables in a database) to be

Email addresses: pietro.pinoli@polimi.it (Pietro Pinoli), stefano.ceri@polimi.it (Stefano Ceri), davide.martinenghi@polimi.it (Davide Martinenghi), luca.nanni@polimi.it (Luca Nanni)

loaded for the query process. Metadata associated with them can be used at query execution time for *selectively* loading data from the relevant repositories and transferring them to the query processing components. Such components can, e.g., be the nodes of a cloud computing system running dedicated database engines.

- The query language must allow for metadata-aware query processing: metadata should thus be enriched as the operations are applied to samples and should describe the partial as well as the final results produced by the query.

However, GMQL was tailored to meet the requirements of biological data and was based on several domain-specific assumptions that only apply when observations refer to DNA regions.

The aim of this paper is, instead, to widen the applicability of metadata-aware query languages by providing a general-purpose approach that encompasses arbitrary observations of scientific databases. To this end, we present the new Scientific Query Language ScQL. We provide a formal definition of the language semantics, which makes it possible to explain the interplay between metadata and observations during the evaluation of ScQL operations and the propagation of metadata to partial and final results. The precise definition of ScQL semantics also enables us to define an important optimization principle, called *meta-first*, that allows executing the computation of metadata operations before loading the observation samples, thereby offering remarkable efficiency improvements.

ScQL supports a new notion of *provenance*: in its classic use, this term denotes the ability to understand, for each observation in the result, which individual observations of the input have contributed to its generation; we refer to this notion as *fine-grained database provenance*, associated to each database tuple. In our approach, datasets produced as query result are decomposed into homogeneous samples, and the observations within each sample are globally described by metadata. Metadata of each result sample are either properties of the input samples that contributed to its construction or aggregate properties computed during query processing. Hence, our approach provides a more global notion of *sample provenance*, which is easier to compute than fine-grained database provenance and yet provides very relevant information to the data scientist.

Sample provenance can be very effective for follow-up activities of data analysis and mining applied to query results; for instance, resulting samples can be classified or clustered by using as parameters, in addition to their values, also their metadata.

From an implementation point of view, metadata and observation computations can take place separately, possibly using very different implementation strategies and even technologies, as metadata are several orders of magnitude smaller than observations. In this paper, we do not discuss how ScQL should be mapped to a physical representation and therefore we cannot discuss its physical optimization.

However, we define the *metadata-first* optimization, a representation independent optimization based on the simple idea of anticipating the computation of metadata over observations whenever possible. The metadata-first optimization is fully original, it emphasizes the use of metadata in order to simplify the queries in a way that can give dramatic advantages and it is formally defined and proved correct. We believe that the meta-first optimization may inspire a class of optimization methods based on similar abstractions.

2. Scientific Data Model

A scientific database is based on the notions of datasets and samples. Datasets are collections of samples, and each sample consists of two parts: the *observations*, which describe specific scientific facts or events, and the *metadata*, which describe general properties of the sample.

2.1. Motivation

Experimental data have a variety of file formats; we just assume them to be self-described through a schema, as advocated in Jim Gray's interview reported in the Fourth Paradigm book [4]. The distinguishing aspect of our approach is that experimental data are associated with metadata describing their provenance, and metadata are processed together with experimental data, so as to propagate the provenance to the query results. Due to the lack of agreed standards for metadata, they are modeled as free attribute-value pairs; we expect metadata to include at least the experiment type, the data collection or analysis method used for data production, and then domain-specific aspects.

2.2. Samples and datasets

We define a *sample* s as a triple of the following form:

$$s = \langle j, \mathcal{M}, O \rangle \tag{1}$$

where:

- j is the sample identifier, distinct for every sample. This is also denoted $id(s)$.
- $\mathcal{M} \subset \mathcal{A} \times \mathcal{V}$ is the *metadata* set; both $\mathcal{A}$ (attributes) and $\mathcal{V}$ (values) are unconstrained sets of strings. The metadata set of s is also denoted $meta(s)$.
- $O \subseteq \mathcal{X}$ is the *observation* set, i.e., a set of vectors in a vector space $\mathcal{X} = \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_n$, where every dimension $\mathcal{X}_i$ is a set of floating point numbers, integers or strings. Set O is also denoted $obs(s)$. As customary, every attribute $\mathcal{X}_i$ is associated with a distinct *attribute name* A_i. The sequence $\langle A_1, \ldots, A_n \rangle$ is called the *schema* of the sample, also indicated as $\texttt{Schema}(s)$. Given an observation $o = \langle x_1, x_2, \ldots, x_n \rangle \in O$, we write $o[A_i]$ to denote x_i, i.e., the value of the attribute labeled by A_i in o. [1]

[1] Both in observations and metadata, any type of data for which properties of equality and relative ordering between instances are defined can be used. For the sake of simplicity, we restricted to basic types for which such properties are naturally defined.

A *dataset* is a set of samples over the same vector space and schema. In the following, we assume that, for every dataset D, metadata and observations are stored in separate structures, $D^{\mathcal{M}} = \langle j, \mathcal{M} \rangle$ and $D^{O} = \langle j, O \rangle$, respectively. Each sample $s = \langle j, \mathcal{M}, O \rangle$ can be simply reconstructed by joining upon the identifier j.

2.3. Examples

Every dataset is typically produced within the same project, by using the same technology and tools, but with different experimental conditions, described by metadata; for instance, a dataset may describe rainfalls observed at a variety of locations, or the tweets extracted for different input hashtags, or genomic data collected at a genomic research center or within an international consortium.

Each dataset is represented by using two tables, one for observations and one for metadata. Observations have an arbitrary schema, but they must include a sample identifier. Metadata are simple triples that add the sample identifier to an attribute-value pair. Identifiers provide a many-to-many relationship between observations and metadata of each sample.

Id	Date	Time	Rainfall
1	12-3-2016	0:17	0.002
1	12-3-2016	0:18	0.004
1	12-4-2016	3:19	0
1	12-4-2016	3:20	0
2	12-3-2016	0:17	0
2	12-3-2016	0:18	0.001
2	12-4-2016	3:19	0
2	12-4-2016	3:20	0
3	12-3-2016	0:17	0.001
3	12-3-2016	0:18	0.004
3	12-4-2016	3:19	0
3	12-4-2016	3:20	0

Id	Attribute	Value
1	region	Tuscany
1	city	Florence
1	device	R23-Meter
2	region	Lombardy
2	city	Bormio
2	device	R45
3	region	Tuscany
3	city	Pisa
3	device	R24-Meter

Table 1: Observations and metadata of a rainfall dataset.

Tables 1 and 2 contain two datasets: the first one is related to *rainfalls* while the second one to *snowfalls*. In both datasets we have as metadata the region and city where the measurement take place and also the identifier of the meter. The observations are temporal series of values of rainfall and snowfall. The temporal coordinates are the date and the time of measurement.

Id	Date	Time	Snowfall
1	11-3-2016	15:00	0.2
1	12-3-2016	15:00	0.3
2	11-3-2016	15:00	0
2	12-3-2016	15:00	0.1
3	11-3-2016	15:00	0.2
3	12-3-2016	15:00	0.2

Id	Attribute	Value
1	region	Tuscany
1	city	Florence
1	device	S10
2	region	Lombardy
2	city	Bormio
2	device	S50
3	region	Tuscany
3	city	Pisa
3	device	S7

Table 2: Observations and metadata of a snow level dataset.

Id	Account	Date	Time	Text
1	@tom	12-3-2016	0:17	memories of #Aprica
1	@bob	12-3-2016	0:20	good sky #Aprica
1	@alice	12-3-2016	0:17	red sunset #Aprica
2	@billy	12-4-2016	3:17	hot water of #Bormio
2	@bob	12-4-2016	3:18	#Bormio forever
2	@alice	12-4-2016	3:21	#Bormio hard slopes

Id	Attribute	Value
1	hashtag	#Aprica
1	date	12-3-2016
1	time	0:21
1	max	300
2	hashtag	#Bormio
2	date	12-4-2016
2	time	3:25
2	max	500

Table 3: Observations and metadata of a tweet dataset.

An example of *twitter dataset* is shown in Table 3; note that each tweet is extracted by a specific access using the Twitter API, hence a Twitter access corresponds to a sample, individual tweets correspond to observations within each sample (with their handle, date, time, text), metadata include the query hashtag, date, time of query submission and max number of retrieved tweets.

Finally, an example of *genomic dataset* is shown in Table 4. Each sample corresponds to an experiment; observations are genomic regions, having specific coordinates (the chromosome where the region belongs, the region's left and right end, the strand describing the direction of DNA reading) and a specific measure (the `P_value` representing how significant a peak of

Id	Chr	Left	Right	strand	Pvalue
1	1	345656	454676	+	0.000024
1	1	3467	45446	-	0.000053
1	3	895656	914676	-	0.000013
1	3	1345656	1454676	+	0.000024
1	11	36667	45555	-	0.000021
2	1	1345656	1454676	+	0.000034
2	1	346777	465446	-	0.000023
2	3	85656	91676	-	0.000033
2	3	5656	14476	+	0.000024
2	14	85656	91476	-	0.000013
2	17	124	455	-	0.000021
2	17	8556	9176	-	0.000016

Id	Attribute	Value
1	cell	CLL
1	tissue	blood
1	sex	F
1	disease	cancer-brca
1	disease	diabetes
2	cell	CLL
2	tissue	blood
2	disease	cancer-coad

Table 4: Observations and metadata of a genomic dataset.

expression is in that genomic region). Metadata typically include the cell line, tissue, experimental condition (e.g., antibody target) and organism sequenced; in case of clinical studies, individual's descriptions including phenotypes. Note that sample 1 has 5 observations and 4 metadata attributes, sample 2 has 7 observations and 3 metadata attributes. Attributes may have multiple values (e.g., the `Disease` attribute can have both values `'Cancer'` and `'Diabetes'`). Throughout the next section we will use genomic examples.

3. Scientific Query Language

The Scientific Query Language (ScQL) is an extension of Relational Algebra for expressing scientific computation and dealing explicitly with metadata management.

3.1. Rationale of Language Design

A scientific query (or program) is expressed as a sequence of operations with the following structure:

```
<variable> = operation(<parameters>) <variables>
```

where each variable stands for a dataset. ScQL operations form a *closed algebra*: results are expressed as new datasets derived from their operands. Operations are either unary (with one input variable) or binary (with two input variables), and construct one result variable. The unary operations are: `SELECT`, `SEMIJOIN`, `PROJECT`, `EXTEND`, `ORDER`, `MERGE`, and `GROUP`; the binary operations are: `UNION`, `DIFFERENCE`, and `JOIN`. They are extensions of classic Relational Algebra operations; domain-specific operations may be added to serve specific domain needs, e.g., in genomics [5].

Compared with languages which are currently in use by the scientific community, ScQL is *declarative* (it specifies the structure of the results, leaving its computation to each operation's implementation) and *high-level* (each operation typically substitutes for a long program that embeds calls to scientific computations)[2]. The progressive computation of variables resembles other algebraic languages (e.g., *Pig Latin* [6]).

Metadata are typically used for selecting the samples of interest from larger datasets, but the distinguishing aspect of ScQL is that metadata are then processed together with experimental data. In many cases, metadata computation is implicit and consists in their transfer from the input variable(s) to the output variable. However, the language allows one to override the implicit metadata management and to explicitly deal with their projection, join, grouping, and sorting. It is important to realize that implicit metadata management is made possible by the scientific data model, which links metadata and observations.

Operators are carefully chosen for an effective and complete expression of relational computations over complex dataset objects formed by observations and metadata, in which each dataset object is further organized as collection of samples and the metadata and observation parts of a sample share the same sample identifier.

Although the design of operations descends from classic Relational Algebra, some operations introduce features that are very specific of our data model, which links observations and metadata. Among them, `EXTEND` computes aggregate properties of observations for each sample and stores them in the sample metadata; `GROUP` creates new samples as the aggregation of those samples with identical values of the grouping attributes.

An important aspect of the language is the implicit management of the sample identifiers, achieved by OID invention in the operations of `MERGE` and `JOIN` and by a careful workflow with information passing between the table-level operations.

Additional design principles of the language are *relational completeness* and *orthogonality* (see Section 3.13 for more details).

Motivational Example (Sketch)

Consider the two dataset shown in Figure 1 and Figure 2. In particular let's call the *rainfall* dataset `RAINDS` and the *snowfall* dataset `SNOWDS`.

We now show an example where we compute the average rainfall and maximum snowfall for a given region and day, we join on the city being observed and we extract the top 100 observations in decreasing values of average rainfall.

```
DS1 = SELECT[region='Tuscany';date=12/3/16] RAINDS
DS2 = SELECT[region='Tuscany';date=12/3/16] SNOWDS
DS3 = EXTEND[sumrain as sum(rainfall)] DS1
```

[2]However, note that within the database community algebraic languages are considered as procedural while calculus and SQL are considered as declarative.

```
DS4 = EXTEND[maxsnow as max(snowlevel)] DS2
DS5 = JOIN [city] DS3, DS4
RESULT = ORDER[DESC LEFT.sumrain; TOP 100] DS5
```

In the next sections the semantics and exact signature of the ScQL operators used in this example will be given. As for now we can anticipate that the SELECT operator enables to filter a dataset based on a condition on metadata and also observations. On the other hand, the EXTEND operator builds a new metadata attribute (sumrain and maxsnow) based on an aggregation on a specific attribute of the observation data. The JOIN operator has a semantics similar to the SQL join and extends it with the possibility on joining on the values of metadata (city). The ORDER operation has an intuitive semantics and signature: it will order the samples of the resulting dataset by adding a metadata attribute order to each of them specifying their ranking.

In Figure 5 the resulting dataset RESULT is shown.

We have one sample for each city and the metadata of the result indicate the exact location of meters and we can correlate, within given cities, the heaviest rain falls and actual flooding of the city spots with snow levels at the meters surrounding the city.

3.2. Rationale of Language Specification

The specification of each operation applies to observations and to metadata separately: the observation part of an operation computes the result observation, the metadata part of the operation computes the associated metadata. Identifiers preserve the many-to-many mapping of observations and metadata.

The logical separation, in every dataset D, between metadata and observations in two different structures, $D^{\mathcal{M}}$ and $D^{\mathcal{O}}$, makes the language specification useful also for ScQL processing: metadata are several orders of magnitude smaller than observations, and therefore we assume that implementations of ScQL will use different data storage and management technologies for them.

The semantics of ScQL operations is described by means of *table-level operations*, e.g., simpler operations that act either upon metadata ("M" suffix) or upon observations ("O" suffix). They are listed in Figure 1, showing the mapping from ScQL operations to table-level operations. Thus, the semantics of every ScQL operation is obtained as the composition of two separate aspects: the *workflow*, which describes how several table-level operations interact, and then the *semantics* of each individual table-level operation.

This choice allows us to use the semantics directly for driving our implementation, whose upper layers consist of a translator mapping ScQL to a workflow of table-level operations. The workflow can be optimized independently of a mapping to a physical engine, and indeed we define a powerful optimization, called *meta-first*, consisting in a graph rewriting, which separates metadata operations from observation operations; the optimization is applicable to most queries and is always beneficial, although the actual benefit is data-dependent and becomes known at execution time. Table-level operations must instead be translated into engine-specific implementations; hence factoring them is highly beneficial - factoring applies to few cases due to language orthogonality but is still useful.

ScQL operation	Table-level operations
SELECT	SelectM, PurgeO, SelectO
SEMIJOIN	SemiJoinM, PurgeO
PROJECT	ProjectM, ProjectO
EXTEND	ExtM, BuildO
ORDER	OrderM, OrderO
GROUP	GroupM, AggrM, AggrO
MERGE	GroupM, MergeM, MergeO
UNION	UnionM, UnionO
JOIN	MatchM, JoinM, JoinO
DIFFERENCE	MatchM, DiffO

Figure 1: ScQL Operations and Table-level operations.

We next proceed with the definition of ScQL operations.

3.3. Selection

Selections use predicates on both metadata and observations, built by arbitrary Boolean expressions of simple predicates, as it is customary in Relational Algebra. They have a different interpretation here.

- Predicates on metadata have an existential interpretation over samples: they select the entire sample if it contains attributes such that the predicate evaluation on their values is true. A predicate can refer to any attribute, even one that is not present in the metadata; when the attribute is missing in a sample, the predicate evaluates to false.
- Predicates on observations have a classical interpretation: they select the observations where the predicate is true. Predicates must use the attributes in the observation's schema, else the operation is illegal.

3.3.1. Syntax

$$D_{out} = \texttt{SELECT}([pm][; po])\, D_{in} \tag{2}$$

The syntax refers to two selection predicates (pm for the metadata and po for the observations), and allows either to be missing, but not both. Both predicates are Boolean parenthesized expressions formed with the logical operators AND, OR and NOT, and atoms of the form $a \circ v$, where a is an attribute, v is a value, and $\circ \in \{=, \neq, <, \leq, >, \geq\}$. As customary, here and in the other operators, $*$ refers to the full list of attributes.

3.3.2. Workflow

The predicate pm is evaluated by the SelectM operation, thereby returning a set $\mathcal{J}$ of identifiers of the selected samples. These are used by PurgeO to filter observation samples. Then, SelectO selects the observations that satisfy the predicate po. Figure 2 illustrates this workflow.

Id	Left.Date	Left.Time	Left.Rainfall	Right.Date	Right.Time	Right.Snowfall
100	12-3-2016	0:17	0.002	12-3-2016	15:00	0.3
100	12-3-2016	0:18	0.004	12-3-2016	15:00	0.3
101	12-3-2016	0:17	0.001	12-3-2016	15:00	0.2
101	12-3-2016	0:18	0.004	12-3-2016	15:00	0.2

Id	Attribute	Value
100	Left.region	Tuscany
100	Left.city	Florence
100	Left.device	R23-Meter
100	Left.sumrain	0.006
100	Right.region	Tuscany
100	Right.city	Florence
100	Right.device	S10
100	Right.maxsnow	0.3
100	order	1
101	Left.region	Tuscany
101	Left.city	Pisa
101	Left.device	R24-Meter
101	Left.sumrain	0.005
101	Right.region	Tuscany
101	Right.city	Pisa
101	Right.device	S7
101	Right.maxsnow	0.2
101	order	2

Table 5: Resulting dataset `RESULT` of the motivational example query

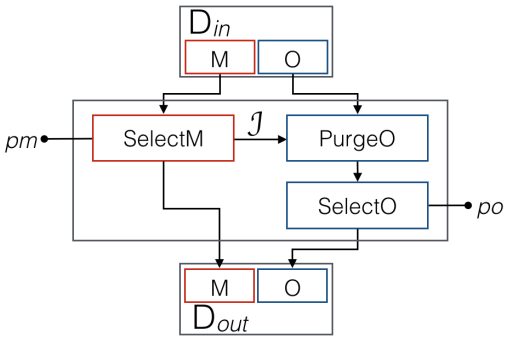

Figure 2: Workflow for `SELECT`.

3.3.3. Semantics

We write $p(\mathcal{M})$ to indicate that predicate p holds when evaluated against a metadata set $\mathcal{M}$. When p is an atom of the form $a \circ v$ we have:

$$p(\mathcal{M}) \text{ iff } \exists v' \mid \langle a, v' \rangle \in \mathcal{M} \wedge v \circ v'.$$

The usual semantics of Boolean operators is used to extend to non-atomic expressions including `AND`, `OR` and `NOT`.

`SelectM` applies the predicate pm to the metadata of D_{in}. The output of `SelectM` is the set $\mathcal{J}$ of identifiers of samples of D_{in} satisfying pm, i.e.:

$$\mathcal{J} = \{j \mid \exists \mathcal{M}\ \langle j, \mathcal{M} \rangle \in D_{in}^{\mathcal{M}} \wedge pm(\mathcal{M})\}.$$

The corresponding set of selected metadata sets, along with their identifiers, is:

$$D_{out}^{\mathcal{M}} = \{\langle j, \mathcal{M} \rangle \mid \ \langle j, \mathcal{M} \rangle \in D_{in}^{\mathcal{M}} \wedge j \in \mathcal{J}\}.$$

`PurgeO` receives the set $\mathcal{J}$ of identifiers extracted by `SelectM` and outputs the corresponding sets of observations $\boldsymbol{O}$ of D_{in} having those identifiers:

$$\boldsymbol{O} = \{\langle j, O \rangle \mid \ \langle j, O \rangle \in D_{in}^{O} \wedge j \in \mathcal{J}\}$$

`SelectO` applies the predicate po to the observations of $\boldsymbol{O}$. Each atomic predicate p of the form $a \circ v$ is evaluated on a given observation o as follows:

$$p(o) \text{ holds iff } o[a] \circ v.$$

As before, the usual semantics of Boolean operators is used to extend to non-atomic expressions including `AND`, `OR` and `NOT`. Let us, for a set of observations O and an observation predicate p, use the classical relational algebra notation $\sigma_p O$ to indicate the set of observations in O satisfying p, i.e., $\sigma_p O = \{o \mid o \in O \wedge p(o)\}$.

The resulting sets of observations, along with their identifiers, are obtained as follows:

$$D_{out}^{O} = \{\langle j, \sigma_{po} O \rangle \mid \langle j, O \rangle \in \boldsymbol{O}\}$$

The dataset D_{out} is the combination of the metadata sets of $D_{out}^{\mathcal{M}}$ and the sets of observations of D_{out}^{O}

$$D_{out} = \{\langle j, \mathcal{M}, O \rangle \mid \langle j, O \rangle \in D_{out}^{O} \wedge \langle j, \mathcal{M} \rangle \in D_{out}^{\mathcal{M}}\}. \quad (3)$$

The combination of the metadata part $D^{\mathcal{M}}_{out}$ and the observation part $D^{\mathcal{O}}_{out}$ will not be shown for the remaining ScQL operations, as it can be performed exactly as in (3).

Overall, the semantics of SELECT, as specified in (2), is captured by the following expression:

$$D_{out} = \{\langle j, \mathcal{M}, \sigma_{po}\mathcal{O}\rangle \mid \langle j, \mathcal{M}, \mathcal{O}\rangle \in D_{in} \wedge pm(\mathcal{M})\}. \quad (4)$$

3.3.4. Example

All examples of unary operations (from Section 3.3 to Section 3.10), including the dataset D_1 in Table 6, use an input schema consisting of five attributes:

$$\texttt{Schema}(D_1) = \texttt{(Chr,Left,Right,Strand,Value)}.$$

The following SELECT produces as output the dataset D_2 in Table 7:

$$D_2 = \texttt{SELECT(antibody = "CTCF"; Value} < 0.25)D_1$$

In all examples with the exception of projection (Section 3.5) the schema of the result is not changed.

id	meta	observations
1	antibody,BRD4 antibody,CTCF organism,HG19	chr1,10,100,*,0.1 chr1,20,300,*,0.5 chr2,250,300,+,0.3 chr3,20,30,*,0.2
2	antibody,CTCF organism,HG19	chr1,40,90,*,0.4 chr1,20,300,*,0.5 chr2,250,300,+,0.1
3	antibody,BRD4 organism,HG19	chr2,25,30,+,0.1 chr3,20,40,+,0.2 chr2,50,300,+,0.5

Table 6: Dataset D_1.

id	meta	observations
1	antibody,BRD4 antibody,CTCF organism,HG19	chr1,10,100,*,0.1 chr3,20,30,*,0.2
2	antibody,CTCF organism,HG19	chr2,250,300,+,0.1

Table 7: Dataset D_2.

The next example illustrates what happens when the metadata of a sample matches, but none of the observations matches the observation predicate.

$$D_3 = \texttt{SELECT(antibody = "BRD4"; Left} > 100)D_1$$

The resulting output D_3 is shown in Table 8, and consists of two samples, one of which contains no observations.

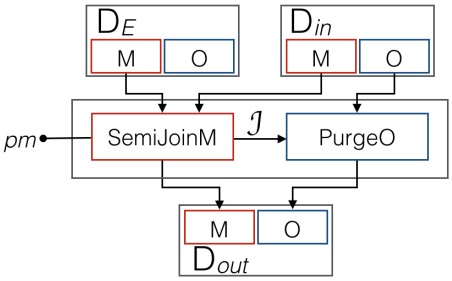

Figure 3: Workflow for SEMIJOIN

id	meta	observations
1	antibody,BRD4 antibody,CTCF organism,HG19	chr2,250,300,+,0.3
3	antibody,BRD4 organism,HG19	

Table 8: Dataset D_3.

3.4. Semi-join

The semi-join operation is also used to select the samples of a dataset, based on an equi-join predicate on its metadata referring to another dataset (called *external dataset*). [3]

3.4.1. Syntax

$$D_{out} = \texttt{SEMIJOIN}(\mathcal{A}_m; D_E)D_{in}. \quad (5)$$

Here, D_{in} and D_E are datasets and $\mathcal{A}_m$ is a list of attributes appearing in the metadata of both D_E and D_{in}.

3.4.2. Workflow

The attribute list $\mathcal{A}_m$ is used by the SemiJoinM operation to return a set $\mathcal{J}$ of identifiers of those samples of D_{in} that, for each metadata attribute in $\mathcal{A}_m$, match the corresponding value in the external dataset D_E. These identifiers are used by PurgeO to filter observation samples. Figure 3 illustrates this workflow.

3.4.3. Semantics

SemiJoinM applies to D_{in} by retaining only those metadata sets that agree with a metadata set in D_E on all attribute-value pairs for the attributes in $\mathcal{A}_m$. Let us write $\mathcal{A}_m(\mathcal{M})$ to indicate that the metadata set $\mathcal{M}$ has the above property:

$$\mathcal{A}_m(\mathcal{M}) \text{ iff } \forall a \in \mathcal{A}_m\, \exists s \in D_E\, \exists v\, \langle a, v\rangle \in \mathcal{M} \cap meta(s).$$

[3] Although SEMIJOIN uses a second dataset D_E, we treat it as a unary operator, because the result is strictly contained in the operand dataset D_{in}. Also, only the metadata of D_E are used, while its observations are completely disregarded; with binary operators, instead, both inputs are used in full.

The result D_{out} will contain a subset of the samples of D_{in}, whose identifiers are:

$$\mathcal{J} = \{j \mid \exists \mathcal{M}\ \langle j, \mathcal{M}\rangle \in D_{in}^{\mathcal{M}} \wedge \mathcal{A}_m(\mathcal{M})\}.$$

The corresponding set of selected metadata sets, along with their identifiers, is:

$$D_{out}^{\mathcal{M}} = \{\langle j, \mathcal{M}\rangle \mid \ \langle j, \mathcal{M}\rangle \in D_{in}^{\mathcal{M}} \wedge j \in \mathcal{J}\}.$$

`PurgeO` receives the identifiers in $\mathcal{J}$ and retains the observations of D_{in} having those identifiers:

$$D_{out}^{O} = \{\langle j, O\rangle \mid \ \langle j, O\rangle \in D_{in}^{O} \wedge j \in \mathcal{J}\}$$

Overall, the semantics of `SEMIJOIN` is captured by the following expression:

$$D_{out} = \{\langle j, \mathcal{M}, O\rangle \mid \langle j, \mathcal{M}, O\rangle \in D_{in} \wedge \mathcal{A}_m(\mathcal{M})\}. \quad (6)$$

3.4.4. Example

Consider again dataset D_1 from Table 6 and a dataset D_3 consisting of one sample with the metadata pairs `<antibody, CTCF>`, `<organism, HG19>`.

The operation

$$D_2 = \texttt{SEMIJOIN(organism)}D_1\ D_3$$

produces in D_2 all the samples of D_{in}.

The operation

$$D_2 = \texttt{SEMIJOIN(organism, antibody)}D_1\ D_3$$

produces in D_2 the samples with identifiers 1 and 2 of D_1.

3.5. Projection

It is used to project metadata and observation attributes.[4] It can also be used to build new attributes as scalar expressions (e.g., for metadata, the age from the birth date; for observations, the length of a region as the difference between its right and left ends).

3.5.1. Syntax

$$D_{out} = \texttt{PROJECT}(am_1[\texttt{AS}\ f_1], \ldots, am_m[\texttt{AS}\ f_m];\\ ao_1[\texttt{AS}\ g_1], \ldots, ao_n[\texttt{AS}\ g_n])D_{in} \quad (7)$$

where the am_i's denote metadata attribute names, the ao_i's denote observation attribute names, and the f_i's and g_i's denote scalar functions computed by using parenthesized mathematical expressions built from attribute names, constant values and standard arithmetic operations.

3.5.2. Workflow

The projection is independently applied to metadata and to observations (with `ProjectM` and `ProjectO`, respectively). Figure 4 illustrates this workflow.

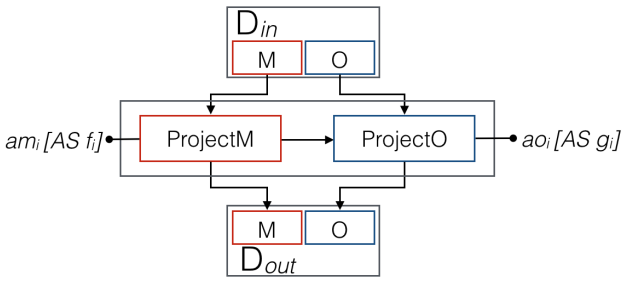

Figure 4: Workflow for `PROJECT`

3.5.3. Semantics

`ProjectM`. Let $\mathcal{A}_m^{new}$ be the set of metadata attribute names that are syntactically associated by the `PROJECT` operation with a scalar function. For each $a_j \in \mathcal{A}_m^{new}$, let f_j be the function associated with a_j and let $f_j(\mathcal{M})$ indicate the value obtained by applying f_j on the metadata set $\mathcal{M}$ [5]. Let $\mathcal{A}_m^{old}$ be the set of the other attribute names in the `PROJECT` operation. Then, for every sample of D_{in} with metadata $\mathcal{M}$, D_{out} will contain a corresponding sample with metadata $\tau_m(\mathcal{M})$, i.e.:

$$D_{out}^{\mathcal{M}} = \{\langle j, \tau_m(\mathcal{M})\rangle \mid \ \langle j, \mathcal{M}\rangle \in D_{in}\},$$

where $\tau_m(\mathcal{M})$ is defined as follows:

$$\tau_m(\mathcal{M}) = \{\langle a, v\rangle \mid \langle a, v\rangle \in \mathcal{M} \wedge a \in \mathcal{A}_m^{old}\} \cup \\ \{\langle a_j, f_j(\mathcal{M})\rangle \mid a_j \in \mathcal{A}_m^{new}\}.$$

`ProjectO`. If observation attribute name ao_i, $1 \le i \le n$, is associated with a scalar function g_i in the `PROJECT` operation, we let $g_i(o)$ indicate the value obtained by applying g_i on the observation o; otherwise, we let $g_i(o)$ indicate the value $o[ao_i]$ associated with attribute ao_i in observation o.

Then, for every sample of D_{in} with observation set O, D_{out} will contain a corresponding sample with observation set $\tau_o(O)$, i.e.:

$$D_{out}^{O} = \{\langle j, \tau_o(O)\rangle \mid \ \langle j, O\rangle \in D_{in}\},$$

where $\tau_o(O)$ is defined as follows:

$$\tau_o(O) = \{\langle g_1(o), \ldots, g_n(o)\rangle \mid o \in O\}.$$

3.5.4. Example

Consider the dataset D_3 in Table 9. The following `PROJECT`:

$$D_4 = \texttt{PROJECT(antibody, count, avg AS total/count;}\\ \texttt{Chr, Left, Right, Length as (Right - Left))}\ D_3$$

produces as output the dataset D_4 in Table 10, with schema:

$\texttt{Schema}(D_4) = (\texttt{Chr}, \texttt{Left}, \texttt{Right}, \texttt{Length})$

[4] A syntactic variant (using the keywords `ALL BUT`) allows one to specify only the attributes that are removed from the result; this variant is very useful with datasets having hundreds of metadata.

[5] When a scalar function is applied to an attribute with either missing or multiple values, its result is undefined and no corresponding metadata attribute is generated.

id	meta	observations
1	antibody,BRD4	chr1,10,100,*,0.1
	antibody,CTCF	chr1,20,300,*,0.5
	total,1.1	chr2,250,300,+,0.3
	count,4	chr3,20,30,*,0.2
2	antibody,CTCF	chr1,40,90,*,0.4
	organism,HG19	chr1,20,300,*,0.5
	total,1.0	chr2,250,300,+,0.1
	count,3	
3	antibody,BRD4	chr2,25,30,+,0.1
	organism,HG19	chr3,20,40,+,0.2
	total,0.8	chr2,50,300,+,0.5
	total,0.9	
	count,3	

Table 9: Dataset D_3.

id	meta	observations
1	antibody,BRD4	chr1,10,100,90
	antibody,CTCF	chr1,20,300,280
	count,4	chr1,250,300,50
	avg,0.275	chr1,20,30,10
2	antibody,CTCF	chr1,40,90,50
	count,3	chr1,20,300,280
	avg,0.333	chr2,250,300,50
3	antibody,BRD4	chr3,25,30,5
	count,3	chr3,20,40,20
		chr2,50,300,250

Table 10: Dataset D_4.

3.6. Extension

It generates new metadata as the result of aggregate functions applied to observation attributes.

3.6.1. Syntax

$$D_{out} = \texttt{EXTEND}(am_1\ \texttt{AS}\ f_1, \ldots, am_n\ \texttt{AS}\ f_n)D_{in} \quad (8)$$

where the am_i's denote metadata attribute names and the f_i's denote aggregate functions computed over parenthesized mathematical expressions built from attribute names, constant values and standard arithmetic operations. Supported aggregate functions include `COUNT` (applicable to any type), `MIN`, `MAX` (applicable to any ordered type, including lexicographically ordered strings) and `SUM`, `AVG`, `MEDIAN`, `STD` (applicable to numeric types).

3.6.2. Workflow

The `BuildO` operation is applied to the observation set of each sample of D_{in} to compute, for each attribute, a corresponding aggregate value. `BuildO` produces such attribute-value pairs (along with the sample identifier); with that, `ExtM` constructs the metadata of the result D_{out}. The observations of D_{in} are kept unchanged in D_{out}. Figure 5 illustrates this workflow.

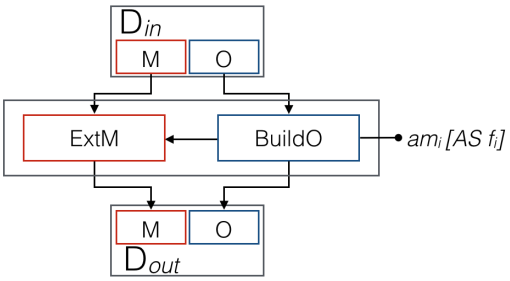

Figure 5: Workflow for `EXTEND`

3.6.3. Semantics

`BuildO` produces the following set of new metadata attribute-value pairs, along with the corresponding sample identifiers:

$$\mathcal{AV} = \{\langle j, am_i, f_i(O)\rangle \mid \langle j, O\rangle \in D^O_{in} \wedge 1 \leq i \leq n\},$$

where $f_i(O)$ indicates the value computed by the aggregate function f_i on the observations O of the sample with identifier j.

`ExtM` builds the metadata sets of D_{out} from $\mathcal{AV}$ and from the metadata sets of D_{in} as follows:

$$D^{\mathcal{M}}_{out} = \{\langle j, \mathcal{M} \cup \tau(j)\rangle \mid \langle j, \mathcal{M}\rangle \in D^{\mathcal{M}}_{in}\},$$

where $\tau(j)$ is the set of attribute-value pairs associated with id j:

$$\tau(j) = \{\langle am, v\rangle \mid \langle j, am, v\rangle \in \mathcal{AV}\},$$

As mentioned, the observations are kept unchanged, i.e.:

$$D^O_{out} = D^O_{in}.$$

3.6.4. Example

Consider the dataset D_5 in Table 11. The following `EXTEND` produces as output the dataset D_6 in Table 12.

$$D_6 = \texttt{EXTEND(avg\_V AS AVG(Value),}$$
$$\texttt{min\_V AS MIN(Value))}\ D_5$$

id	meta	observations
1	antibody,BRD4	chr1,10,100,*,0.1
	antibody,CTCF	chr1,20,300,*,0.5
	organism,HG19	chr2,250,300,+,0.3
		chr3,20,30,*,0.2
2	antibody,CTCF	chr1,40,90,*,0.4
	organism,HG19	chr1,20,300,*,0.5
		chr2,250,300,+,0.1
3	antibody,BRD4	chr2,25,30,+,0.1
	organism,HG19	chr3,20,40,+,0.2
		chr2,50,300,+,0.5

Table 11: Dataset D_5.

id	meta	observations
1	antibody,BRD4	chr1,10,100,*,0.1
	antibody,CTCF	chr1,20,300,*,0.5
	organism,HG19	chr2,250,300,+,0.3
	avg_V,0.275	chr3,20,30,*,0.2
	min_V,0.1	
2	antibody,CTCF	chr1,40,90,*,0.4
	organism,HG19	chr1,20,300,*,0.5
	avg_V,0.333	chr2,250,300,+,0.1
	min_V,0.1	
3	antibody,BRD4	chr2,25,30,+,0.1
	organism,HG19	chr3,20,40,+,0.2
	avg_V,0.266	chr2,50,300,+,0.5
	min_V,0.1	

Table 12: Dataset D_6.

3.7. Order

It is used for ordering samples, observations, or both of them. The default ordering is *ascending*, and can be turned to *descending* by an explicit indication. Sorted samples have a new attribute `Order`, added to metadata, observations, or both of them[6]. The value of `Order` reflects the result of the sorting. If sorting is applied to samples or observations that were previously ordered, such previous ordering is not considered. Identifiers of the samples of the operand D_{in} are assigned to the result. The clause `TOP` k extracts the first k observations or samples.

3.7.1. Syntax

$$D_{out} = \texttt{ORDER}([[\texttt{DESC}]am_1, \ldots, [\texttt{DESC}]am_n[\texttt{TOP } k]] \quad (9)$$
$$[; [\texttt{DESC}]ao_1, \ldots, [\texttt{DESC}]ao_m[\texttt{TOP } h]])D_{in}$$

where the am_i's denote metadata attribute names and the ao_i's denote observation attribute names.

3.7.2. Workflow

The ordering is independently applied to metadata and to observations. When the `TOP` clause is used in the metadata to filter some of the samples, the corresponding identifiers, computed by `OrderM`, are included in the set $\mathcal{J}$ and used by `OrderO`.

3.7.3. Semantics

For a dataset D, let $rank_D(j)$ indicate the rank, between 1 and $|D|$, of the sample identified by j when the samples of D are sorted according to the metadata attributes listed in the `ORDER` operation. Similarly, for an observation set O, let $rank_O(o)$ indicate the rank between 1 and $|O|$ of observation $o \in O$ when the observations of O are sorted according to the observation attributes listed in the `ORDER` operation.

[6] `Order` and `Group` are reserved words that cannot be used as attribute names.

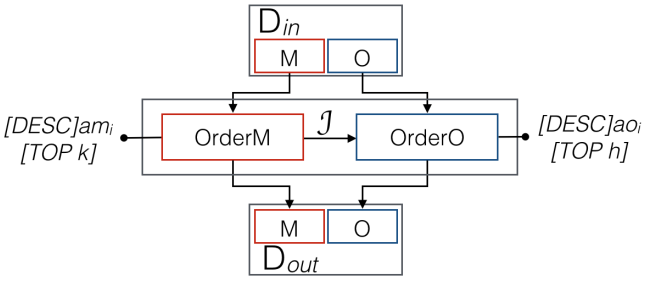

Figure 6: Workflow for `ORDER`.

`OrderM`. Let k be the parameter of the `TOP` k clause (consider $k = +\infty$ if the `TOP` k clause is not present). Then the set of retained sample identifiers is as follows:

$$\mathcal{J} = \{j \mid \exists \mathcal{M}\ \langle j, \mathcal{M}\rangle \in D^{\mathcal{M}} \wedge rank_D(j) \leq k\}$$

The metadata set of each retained sample is extended with an `Order` attribute as follows:

$$D^{\mathcal{M}}_{out} = \{\langle j, \mathcal{M} \cup \{\langle \texttt{Order}, rank_D(j)\rangle\}\rangle \mid \langle j, \mathcal{M}\rangle \in D^{\mathcal{M}}\}$$

`OrderO`. The schema of D_{out} is obtained from the schema of D_{in} by adding the `Order` attribute, i.e.:

$$\texttt{Schema}(D_{out}) = \texttt{Schema}(D_{in}) \cdot \langle \texttt{Order}\rangle$$

where $\cdot$ indicates tuple concatenation. For every observation set O of a sample in D_{in} whose identifier is in $\mathcal{J}$, every observation $o \in O$ is then modified by adding a value $rank_O(o)$ for the `Order` attribute, as specified below, where h is the parameter of the `TOP` h clause (consider $h = +\infty$ otherwise):

$$D^{O}_{out} = \{\langle j, \tau(O)\rangle \mid \langle j, O\rangle \in D^{O} \wedge j \in \mathcal{J}\}$$

where, for an observation set O, $\tau(O)$ is computed as follows:

$$\tau(O) = \{o \cdot \langle rank_O(o)\rangle \mid o \in O \wedge rank_O(o) \leq h\}$$

3.7.4. Example

Consider the dataset D_7 in Table 13. The following `ORDER` produces D_8 in Table 14.

$$D_8 = \texttt{ORDER}(\texttt{Count}; \texttt{Value TOP } 2)D_7$$

id	metadata	observations
1	antibody,CTCF	chr1,10,20,+,0.3
	count,5	chr1,10,20,+,0.1
		chr1,15,20,+,0.5
2	antibody,CTCF	chr1,10,20,+,0.3
	count,3	chr2,10,30,+,0.2
		chr1,15,20,+,0.1

Table 13: Dataset D_7.

id	metadata	observations
1	antibody,CTCF count,5 order,2	chr1,10,20,+,0.3,2 chr1,10,20,+,0.1,1
2	antibody,CTCF count,3 order,1	chr1,15,20,+,0.1,1 chr2,10,30,+,0.2,2

Table 14: Dataset D_8.

3.8. Group

It is used for grouping samples or observations according to distinct values of grouping attributes, and then for computing aggregate values for each group.

- For what concerns metadata, each combination of distinct values of the grouping attributes is associated with one new attribute `Group`, which carries a distinct value for each group; samples having missing values for any of the grouping attributes are discarded. As metadata attributes are multi-valued, samples of the input dataset are partitioned by each subset of their distinct values (e.g., samples with a `Disease` attribute equal both to `'Cancer'` and to `'Diabetes'` are within a group that is distinct from the groups of the samples with only one value, either `'Cancer'` or `'Diabetes'`). New metadata attributes are added to the output samples for storing the results of aggregate function evaluations over each group.
- For what concerns observations, grouping applies to the observations having the same values for the grouping attributes. The resulting schema includes new attributes for storing the results of the evaluation of aggregate functions on each group.

Note that, after a `GROUP` operation, every sample of the input dataset with a legal grouping value corresponds to a sample of the output dataset; the `MERGE` operation, which is next discussed, can then be used to merge all samples belonging to the same group into a single sample.

3.8.1. Syntax

$$
\begin{aligned}
D_{out} = &\texttt{GROUP}(\\
&[am_1, \ldots, am_m[, am'_1\ \texttt{AS}\ f_1, \ldots, am'_{m'}\ \texttt{AS}\ f_{m'}]]\\
&[; ao_1, \ldots, ao_n[, ao'_1\ \texttt{AS}\ g_1, \ldots, ao'_{n'}\ \texttt{AS}\ g_{n'}]])D_{in} \qquad (10)
\end{aligned}
$$

where the am_i's denote metadata attribute names of D_{in} used for grouping, the ao_i's denote observation attribute names used for grouping, the am'_i's denote new metadata attribute names, the ao'_is denote new observation attribute names, the f_i's and g_i's denote aggregate functions used for computing the values corresponding to the new attribute names. Aggregates are defined as in the `EXTEND` operation. For convenience, we let $\mathcal{A}_m = am_1, \ldots, am_m$ indicate the grouping metadata attributes.

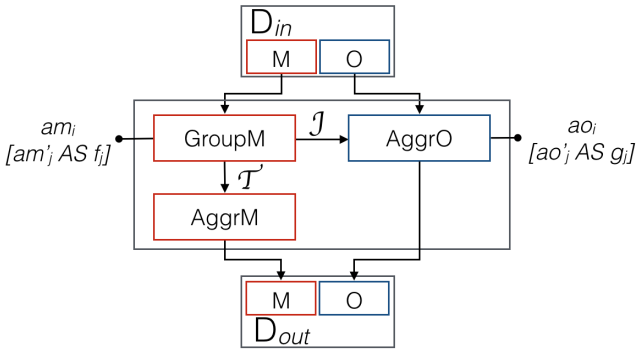

Figure 7: Workflow for `GROUP`.

3.8.2. Workflow

The grouping metadata attributes $\mathcal{A}_m$ are used by the `GroupM` operation to identify a partitioning of D_{in}'s samples. Each partition is identified by a new sample identifier k and associated with the set $\mathcal{J}_k$ of identifiers of the samples of D_{in} that are mapped to k. These identifiers are passed to `AggrM` and to `AggrO` for computing aggregates on metadata and observations. As the grouping on metadata is optional, if it is omitted then the aggregates are directly computed on the initial samples in D_{in}.

3.8.3. Semantics

`GroupM`. Given the grouping metadata attributes $\mathcal{A}_m$ and the metadata sets of D_{in}, `GroupM` creates distinct groups for each grouping value. Two samples s_i and s_j of D_{in} belong to the same group, denoted as $s_i \approx_{\mathcal{A}_m} s_j$, if and only if they have exactly the same set of values for every attribute $am \in \mathcal{A}_m$, i.e.

$$
\begin{aligned}
s_i \approx_{\mathcal{A}_m} s_j \text{ iff}&\\
&\{\langle am, v\rangle \mid \langle am, v\rangle \in meta(s_i) \wedge am \in \mathcal{A}_m\}\\
=&\{\langle am, v\rangle \mid \langle am, v\rangle \in meta(s_j) \wedge am \in \mathcal{A}_m\}.
\end{aligned}
$$

Given this definition, grouping has important properties:

- reflexive: $s_i \approx_{\mathcal{A}_m} s_i$;
- commutative: $s_i \approx_{\mathcal{A}_m} s_j \iff s_j \approx_{\mathcal{A}_m} s_i$;
- transitive: $s_i \approx_{\mathcal{A}_m} s_j \wedge s_k \approx_{\mathcal{A}_m} s_i \iff s_k \approx_{\mathcal{A}_m} s_j$.

Let $\mathcal{G}$ be the partition of D_{in} based on $\mathcal{A}_m$, i.e., the set of sets of samples of D_{in} belonging to the same group based on $\mathcal{A}_m$. Then, the output of `GroupM` is a structure

$$\mathcal{T} = \{\langle id(\mathcal{G}), idg(\mathcal{G})\rangle \mid \mathcal{G} \in \mathcal{G}\}, \qquad (11)$$

with

$$idg(\mathcal{G}) = \bigcup_{s \in \mathcal{G}} id(s), \qquad (12)$$

where $id(\mathcal{G})$ is a new group identifier associated with the partition $\mathcal{G}$; $\mathcal{T}$ associates each group identifier $id(\mathcal{G})$ with the set $idg(\mathcal{G})$ of identifiers of the samples associated with the group.

GroupM also produces the set $\mathcal{J}$ of identifiers of all the samples of D_{in} that belong to some group:

$$\mathcal{J} = \bigcup_{\mathcal{G} \in \boldsymbol{\mathcal{G}}} \{s \mid s \in idg(\mathcal{G})\}$$

AggrM receives as input $\mathcal{T}$ and produces the metadata of the output samples. The metadata set of each retained sample s is first extended with a Group attribute as follows:

$$\boldsymbol{\mathcal{M}} = \{\langle j, \mathcal{M} \cup \{\langle \texttt{Group}, id(\mathcal{G})\rangle\}\rangle \mid \langle j, \mathcal{M}\rangle \in D_{in}^{\mathcal{M}} \wedge j \in idg(\mathcal{G}) \wedge \mathcal{G} \in \boldsymbol{\mathcal{G}}\}.$$

Then, AggrM computes the aggregate functions for metadata; if grouping on metadata is omitted, it directly applies to D_{in}, and all samples are associated with the metadata tuple $\langle \texttt{Group}, 1\rangle$. Let am'_h denote the new metadata attribute that must be computed by an aggregate function f_h, and let v_{kh} be the result of the evaluation of f_h over the k-th group, i.e.

$$v_{kh} = f_h(\mathcal{G}), \text{ where } k = id(\mathcal{G}) \text{ and } \mathcal{G} \in \boldsymbol{\mathcal{G}}.$$

The metadata sets of D_{out} are then built by adding to the metadata sets of $\boldsymbol{\mathcal{M}}$ the new pairs $\langle am'_h, v_{kh}\rangle$ to the samples of the k-th group, for every new metadata attribute am'_h:

$$D_{out}^{\mathcal{M}} = \{\langle j, \mathcal{M} \cup \Delta^{\mathcal{M}}(j)\rangle \mid \langle j, \mathcal{M}\rangle \in \boldsymbol{\mathcal{M}}\},$$

where $\Delta^{\mathcal{M}}(j)$ is the set of new attribute-value pairs to be added to the metadata of the sample with identifier j, i.e.:

$$\Delta^{\mathcal{M}}(j) = \bigcup_{1 \le h \le m'} \{\langle am'_h, v_{kh}\rangle \mid k = id(\mathcal{G}) \wedge \mathcal{G} \in \boldsymbol{\mathcal{G}} \wedge j \in idg(\mathcal{G})\}.$$

AggrO receives as input $\mathcal{J}$ and computes the aggregate functions for the corresponding observations. If grouping on metadata is omitted, it directly applies to D_{in}.

The resulting schema of dataset D_{out} is the concatenation of the grouping observation attributes and the new observation attributes, i.e.:

$$\texttt{Schema}(D_{out}) = \langle ao_1, \ldots, ao_n, ao'_1, \ldots, ao'_{n'}\rangle.$$

The observation sets of D_{out} are computed from those in D as follows:

$$D_{out}^{O} = \{\langle j, \tau_o(O)\rangle \mid \langle j, O\rangle \in D_{in}^{O}\}$$

where $\tau_o(O)$ extends the observations of O with the new aggregate values:

$$\tau_o(O) = \{\langle o[ao_1], \ldots, o[ao_n], g_1(O'), \ldots, g_{n'}(O')\rangle \mid o \in O \wedge O' = \sigma_{ao_1 = o[ao_1] \wedge \ldots \wedge ao_n = o[ao_n]} O\}$$

3.8.4. Examples

Consider the dataset D_9 in Table 15. The following two GROUP operations produce the result datasets D_{10} in Table 16 and D_{11} in Table 17, respectively.

$$D_{10} = \texttt{GROUP(antibody, New AS count(*))} D_9$$
$$D_{11} = \texttt{GROUP(; Chr, Start, Stop, New AS max(Value))} D_9$$

id	metadata	observations
1	antibody,CTCF	chr1,10,20,+,0.1 chr1,10,20,+,0.3
2	antibody,CTCF	chr1,10,20,+,0.1 chr2,20,50,-,0.2
3	antibody,CTCF antibody, POL2	chr2,30,40,+,0.1 chr2,30,40,+,0.4

Table 15: Dataset D_9.

id	metadata	observations
1	antibody,CTCF group,1 new,2	chr1,10,20,+,0.1 chr1,10,20,+,0.3
2	antibody,CTCF group,1 new,2	chr1,10,20,+,0.1 chr2,20,50,-,0.2
3	antibody,CTCF antibody, POL2 group,2 new,1	chr2,30,40,+,0.1 chr2,30,40,+,0.4

Table 16: Dataset D_{10}.

id	metadata	observations
1	antibody,CTCF	chr1,10,20,0.3
2	antibody,CTCF	chr1,10,20,0.1 chr2,20,50,0.2
3	antibody,CTCF antibody,POL2	chr2,30,40,0.4

Table 17: Dataset D_{11}.

3.9. Merge

It generates a dataset consisting of a single sample by merging both metadata and observations. Optionally, it is possible to group samples by distinct values of the grouping attribute, as in the GROUP operation; in such a case, merging applies to each group and produces a single sample for each group.

3.9.1. Syntax

$$D_{out} = \texttt{MERGE}[(am_1, \ldots, am_m)] D_{in}, \quad (13)$$

where $am_1, \ldots, am_m$ are grouping metadata attributes.

3.9.2. Workflow

First, grouping is performed by GroupM, as defined above for GROUP. Then, for each group, MergeM combines in a single metadata set all the metadata sets of the samples in the same group. Similarly, MergeO combines in a single observation set all the observation sets of the samples in the same group.

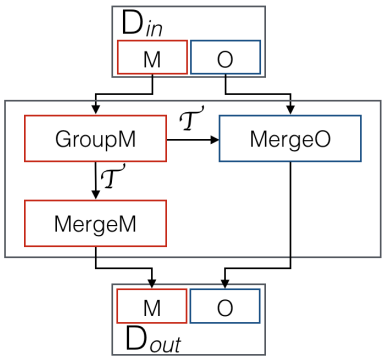

Figure 8: Workflow for MERGE.

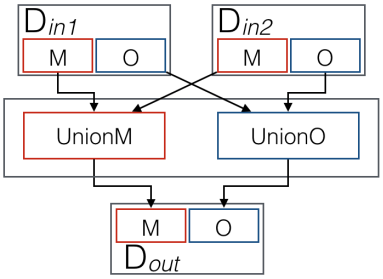

Figure 9: Workflow for UNION.

3.9.3. Semantics

GroupM generates the structure $\mathcal{T} = \langle k, \mathcal{J}_k \rangle$ as in (11), where k is a new sample identifier and $\mathcal{J}_k$ is the set of identifiers of samples of D_{in} that belong to the group identified by k. When no grouping is specified, $\mathcal{T}$ is a singleton.

MergeM creates one metadata set per group:

$$D^{\mathcal{M}}_{out} = \{\langle k, \bigcup_{s \in D_{in},\, id(s) \in \mathcal{J}_k} meta(s) \rangle \mid \langle k, \mathcal{J}_k \rangle \in \mathcal{T}\}$$

MergeO creates one observation set per group:

$$D^{\mathcal{O}}_{out} = \{\langle k, \bigcup_{s \in D_{in},\, id(s) \in \mathcal{J}_k} obs(s) \rangle \mid \langle k, \mathcal{J}_k \rangle \in \mathcal{T}\}$$

3.9.4. Example

Consider the dataset D_9 in Table 15. The following MERGE operation produces the result datasets D_{12} in Table 18.

$$D_{12} = \texttt{MERGE(antibody)}D_9$$

id	metadata	observations
101	antibody,CTCF	chr1,10,20,+,0.1
		chr1,10,20,+,0.3
		chr2,20,50,-,0.2
102	antibody,CTCF	chr2,30,40,+,0.1
	antibody, POL2	chr2,30,40,+,0.4

Table 18: Dataset D_{12}.

3.10. Union

It is used to merge observations of two datasets within a single dataset. Union can be applied only to observations with the same schema; schema differences can be resolved by suitable projections.

3.10.1. Syntax

$$D_{out} = \texttt{UNION}\ D_{in_1}\ D_{in_2} \tag{14}$$

where D_{in_1} and D_{in_2} are datasets with the same schema. The schema of the result D_{out} is also the same, i.e.,

$$\texttt{Schema}(D_{out}) = \texttt{Schema}(D_{in_1}) = \texttt{Schema}(D_{in_2}).$$

3.10.2. Workflow

The operation is independently applied to metadata and to observations.

3.10.3. Semantics

The samples in D_{out} are the union of those in D_{in_1} and D_{in_2}, but the identifiers are reassigned to each sample so that they are still unique within D_{out}. Let $a_1(id(s_1))$ denote the identifier assigned to a sample $s_1 \in D_{in_1}$ and $a_2(id(s_2))$ the identifier assigned to a sample $s_2 \in D_{in_2}$.

UnionM puts together the metadata of the samples of D_{in_1} and D_{in_2} so that the result has the following metadata sets along with the new identifiers:

$$D^{\mathcal{M}}_{out} = \{\langle a_1(j), \mathcal{M} \rangle \mid \langle j, \mathcal{M} \rangle \in D^{\mathcal{M}}_{in_1}\} \cup \{\langle a_2(j), \mathcal{M} \rangle \mid \langle j, \mathcal{M} \rangle \in D^{\mathcal{M}}_{in_2}\}.$$

UnionO takes care of the observations:

$$D^{\mathcal{O}}_{out} = \{\langle a_1(j), O \rangle \mid \langle j, O \rangle \in D^{\mathcal{O}}_{in_1}\} \cup \{\langle a_2(j), O \rangle \mid \langle j, O \rangle \in D^{\mathcal{O}}_{in_2}\}.$$

3.11. Join

The JOIN operation applies to two datasets and acts in two phases. In the first phase, pairs of samples which satisfy the *meta join clause* are identified; the join predicate is an implicit equality comparison among metadata attributes listed as parameters of the operation. In the second phase, observations from matched pairs of samples that satisfy the *observation join clause* are joined. When the first clause is missing, the Cartesian product of samples is performed. When the second clause is missing, the Cartesian product of tuples is performed. The schema of D_{out} is obtained as the concatenation of the schemas of D_{in_1} and of D_{in_2}, i.e.:

$$\texttt{Schema}(D_{out}) = \texttt{Schema}(D_{in_2}) \cdot \texttt{Schema}(D_{in_2}).$$

3.11.1. Syntax

$$D_{out} = \texttt{JOIN}([\mathcal{A}_m][; po])\ D_{in_1}\ D_{in_2} \tag{15}$$

where $\mathcal{A}_m$ is a list of metadata attributes, po is the join predicate for observations, built as a conjunctive expression whose

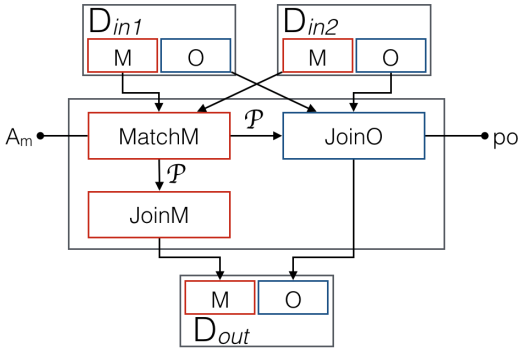

Figure 10: Workflow for JOIN.

atoms are in the form $ao_1 \circ ao_2$, with $ao_1 \in \texttt{Schema}(D_{in_1})$ and $ao_2 \in \texttt{Schema}(D_{in_2})$ and $\circ \in \{=, \neq, <, \leq, >, \geq\}$. Note that a disambiguation prefix may sometimes be prepended to an attribute name; we use "`Left.`" to refer to D_{in_1} and "`Right.`" to D_{in_2}; if nothing is indicated, the left operand refers to D_{in_1}, and the right one to D_{in_2}, as customary in database notation.

3.11.2. Workflow

The `MatchM` operation produces the pairs $\mathcal{P}$ of identifiers of input samples that satisfy the meta join predicate. Such pairs are used by `JoinM` and `JoinO` for building the metadata sets and observation sets of the result.

3.11.3. Semantics

`MatchM` builds the set of pairs of identifiers of the samples of D_{in_1} and D_{in_2} that agree on the metadata attributes in $\mathcal{A}_m$. Therefore $\mathcal{P}$ is built as follows:

$$\mathcal{P} = \{\langle j_1, j_2\rangle \mid \langle j_1, \mathcal{M}_1\rangle \in D^{\mathcal{M}}_{in_1} \wedge \langle j_2, \mathcal{M}_2\rangle \in D^{\mathcal{M}}_{in_2} \wedge \forall am \in \mathcal{A}_m \, \exists v \, (\langle am, v\rangle \in \mathcal{M}_1 \cap \mathcal{M}_2)\}. \quad (16)$$

For each pair $\langle j_1, j_2\rangle$ of identifiers in $\mathcal{P}$, we let ν_{j_1,j_2} denote a new distinct identifier. Note that, when the meta-join clause is omitted, $\mathcal{P}$ simply coincides with the Cartesian product of the identifiers of the samples of D_{in_1} and D_{in_2}.

`JoinM` receives $\mathcal{P}$ and builds the metadata of D_{out}. Then, the metadata sets of D_{out}, along with their new identifiers, are computed as follows:

$$D^{\mathcal{M}}_{out} = \{\langle \nu_{j_1,j_2}, \mathcal{M}_1 \cup \mathcal{M}_2\rangle \mid \langle j_1, j_2\rangle \in \mathcal{P} \wedge \langle j_1, \mathcal{M}_1\rangle \in D^{\mathcal{M}}_{in_1} \wedge \langle j_2, \mathcal{M}_2\rangle \in D^{\mathcal{M}}_{in_2}\}.$$

`JoinO` receives in input the pairs in $\mathcal{P}$ and generates the observation sets of D_{out} by concatenating the observations of D_{in_1} and D_{in_2}:

$$D^{\mathcal{O}}_{out} = \{\langle \nu_{j_1,j_2}, \tau(O_1, O_2)\rangle \mid \langle j_1, j_2\rangle \in \mathcal{P} \wedge \langle j_1, O_1\rangle \in D^{\mathcal{O}}_{in_1} \wedge \langle j_2, O_2\rangle \in D^{\mathcal{O}}_{in_2}\},$$

where $\tau(O_1, O_2)$ takes care of producing the resulting observation set by concatenating the observations that satisfy the observation join clause po:

$$\tau(O_1, O_2) = \{o_1 \cdot o_2 \mid o_1 \in O_1 \wedge o_2 \in O_2 \wedge po(o_1, o_2)\}$$

3.11.4. Example

Consider the input datasets D_L and D_R in Table 19, with a simple schema consisting of `Chrom, Start, Stop`. The following `JOIN` operation produces D_{13} in Table 20:

D_{13} = `JOIN(ant; Chrom = Chrom AND Start > Start)` D_L D_R

with schema:

```
Left.Chrom, Left.Start, Left.Stop,
Right.Chrom, Right.Start, Right.Stop
```

Dataset D_L

id	metadata	observations
1	ant,CTCF	chr1,10,20
	org,HG19	chr1,40,50
		chr2,15,20
2	ant,CTCF	chr1,20,30

Dataset D_R

id	metadata	observations
1	ant,CTCF	chr1,15,25
	dis,cancer	chr2,15,20
2	ant,BRD4	chr1,15,25
		chr2,15,20

Table 19: Datasets D_L and D_R.

id	metadata	observations
201	ant,CTCF	chr1,40,50,chr1,15,25
	org,HG19	
	dis,cancer	
202	ant,CTCF	chr1,20,30,chr1,15,25
	dis,cancer	

Table 20: Dataset D_{13}.

3.12. Difference

This operation applies to two datasets (minuend and subtrahend) with the same schema, and produces one sample in the result for every sample of the first operand (the minuend), with identical identifier and metadata. It considers all the observations of the second operand (the subtrahend), which we denote as *negative observations*, and includes in the corresponding result sample those observations that do not appear as negative observation. When a list of metadata attributes is present, for each sample of the minuend we consider as negative observations only the observations of the samples matching such attributes.

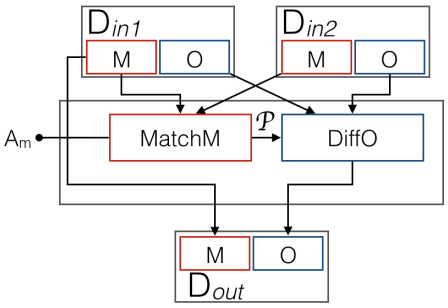

Figure 11: Workflow for DIFFERENCE.

3.12.1. Syntax

$$D_{out} = \texttt{DIFFERENCE}[(\mathcal{A}_m)]D_{in_1}\ D_{in_2} \tag{17}$$

where $\mathcal{A}_m$ is a list of metadata attributes.

3.12.2. Workflow

The MatchM operation builds the set $\mathcal{P}$ of pairs of input samples matching the metadata attributes in $\mathcal{A}_m$. For such pairs, DiffO builds the observations.

3.12.3. Semantics

MatchM builds the set $\mathcal{P}$ of pairs of samples matching the metadata attributes in $\mathcal{A}_m$ as already shown in (16).

DiffO receives in input the pairs in $\mathcal{P}$ and generates the observations of each sample s of D_{out} from the samples of D_{in_1} as follows:

$$D^{O}_{out} = \{\langle j_1, O_1 - \mathcal{N}(j_1)\rangle \mid \langle j_1, O_1\rangle \in D^{O}_{in_1}\}$$

where $\mathcal{N}(j_1)$ is the set of negative observations to be considered for the difference with the sample with identifier j_1:

$$\mathcal{N}(j_1) = \bigcup_{\substack{\langle j_1, j_2\rangle \in \mathcal{P} \wedge \\ \langle j_2, O_2\rangle \in D^{O}_{in_2}}} O_2$$

3.12.4. Example

Consider the input datasets D_L and D_R in Table 19. The following DIFFERENCE statement produces D_{14} in Table 21:

$$D_{14} = \texttt{DIFFERENCE(ant)}\ D_L\ D_R$$

id	metadata	observations
1	ant,CTCF	chr1,10,20
	org,HG19	chr1,40,50
2	ant,CTCF	chr1,20,30

Table 21: Dataset D_{14}.

3.13. Additional design principles

Here we present a discussion on the properties of relational completeness and orthogonality, which we used as driving factors in the design of ScQL.

With relational completeness, we indicate that, although the data model at hand is not itself relational, the classical algebraic manipulations are all supported by simply emulating the notion of relational table through the observation set of a sample. To this end, consider a restricted usage of our model, in which metadata are ignored and each dataset consists of a single sample (and thus its observation set corresponds to a relational table). This suffices to capture the expressive power of classical relational algebra: indeed, when omitting the specification of metadata attributes/predicates in ScQL's SELECT, PROJECT, UNION, DIFFERENCE, and JOIN operators, they capture, respectively, relational algebra's selection, projection, union, difference, and Cartesian product, which are the conventional minimal set of relational operators introduced by Codd [7].[7]

Orthogonality indicates that no operator can be obtained as a combination of other operators. In ScQL, admittedly with the exception of SEMIJOIN (which corresponds to the composition of SELECT and JOIN, but was introduced in order to facilitate query formulation, as customary in many languages), all the operators are orthogonal.

Formally, we say that the (ScQL) operators in a set $\mathcal{S}$ are orthogonal if, for every operator op $\in \mathcal{S}$, there is a ScQL query using op that cannot be expressed in ScQL by only using operators in $\mathcal{S} \setminus \{\texttt{op}\}$.

Theorem 1. *The ScQL operators in* { SELECT, PROJECT, UNION, DIFFERENCE, JOIN, MERGE, UNION, EXTEND, GROUP, ORDER } *are orthogonal.*

Proof. The operators in { SELECT, PROJECT, UNION, DIFFERENCE, JOIN } are orthogonal because the set of relational operators $\{\sigma, \Pi, \cup, -, \bowtie\}$ forms a minimal set of operators [7], and the additional expressive power of the ScQL operators with respect to their relational counterpart only concerns metadata.

The EXTEND operator cannot be expressed by any combination of the other operators because it builds aggregate information about the observations that is then modeled as new metadata information, and this basic operation is not supported by other operators of the language.

MERGE is a unary operation whose effect is to create a single sample out of many samples by making unions of observations and metadata and cannot be expressed by other operations, including UNION which is a binary operation.

The GROUP operator is the only one able to aggregate separately metadata and observation values creating new attributes of both kind with aggregate functions. Differently from MERGE it preserves the same samples as the input.

ORDER is the only operator which builds an order relation between samples or observations. This operation requires the

[7] Technically, however, for relational completeness, one also needs a renaming operator, whose functionality is straightforwardly achieved by ScQL's PROJECT operator.

comparison of observation or metadata values between samples and it is not supported by any of the other operators.

□

4. Optimization of ScQL Queries

We recall from Section 3 that input metadata and observations are stored separately. Metadata have a small size and are stored in a structure $D^{\mathcal{M}} = \langle j, \mathcal{M} \rangle$; they can be implemented by standard technology (e.g., relational) and make use of conventional indexing structures. Observations have a big size and are stored in a structure $D^{O} = \langle j, O \rangle$; we assume D^{O} to be the file system, and j to stand for the file name or identifier. Thus, observations must be selectively loaded into the distributed memory of an execution framework before being used for query computation. This organization is typical of most scientific applications, as observations are stored within global servers and their implementation is based on cloud computing frameworks. To make the interaction of the query with the repository environment more explicit, we add operations `LOAD` and `STORE` to the query.

4.1. Example

A simple example of query Q_1, consisting of two `SELECT` and one `JOIN`, is shown next:

```
LOAD INPUT1;
LOAD INPUT2;
A = SELECT(antibody="CTCF";
           score > 0.4) INPUT1;
B = SELECT(antibody="JUN";
           score > 0.4) INPUT2;
Result = JOIN(CellLine;
              (Left.Start < Right.End AND
               Right.Start < Left.End)) A,B;
STORE Result;
```

In the above query, `INPUT1` and `INPUT2` are the input datasets, collecting observations from a *genomic repository*; the program extracts into the variables `A` and `B` samples that are treated with the `Antibody` equal to `CTCF` and `JUN`, whose observations have a sufficient significance (`Score > 0.4`); then it joins `A` and `B` producing the variable `Result`, returned as query result. The samples of `Result` are paired from the samples of `A` and `B` when the values of the metadata attribute `CellLine` are equal, and the regions of `Result` are composed from the regions of `A` and `B` when the regions of `A` precede the regions of `B` and they intersect.

4.2. Query Translation

The execution of ScQL queries uses *table-level operations* that were introduced in Section 3. The query compiler produces an abstract representation, called *operator DAG*, which describes the precedence between table-level operators; see Figure 12. The figure shows that query optimization can be separated into two parts, a *logical optimization* which applies to the DAG and the *physical optimization* which applies to the execution environment; thus, logical optimization is *independent of*

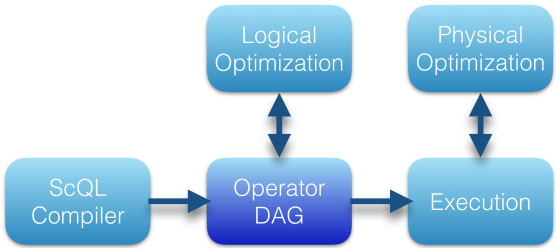

Figure 12: Translation and execution of ScQL.

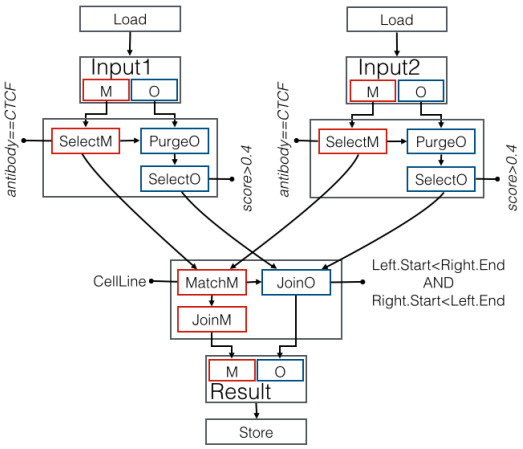

Figure 13: DAG Representation.

the execution environment. Query execution consists of a recursive traversal of the DAG, triggered by the ScQL operation `STORE`; we consider queries with a single `STORE` operation[8].

Figure 13 shows the translation of query Q_1; the figure includes nodes for `LOAD`, `STORE`, and for the variables `INPUT1`, `INPUT2` and `Result`, and then the workflow describing the translation of `SELECT` and `JOIN`, as defined in Sections 3.3 and 3.11. Intermediate variables `A`, `B` are not shown, because `SELECT`'s outputs are directly connected to the `JOIN` inputs. The resulting DAG consists of table-level operations, partitioned into the two sets $\mathcal{D}_{\mathcal{M}}$ of operations on metadata and $\mathcal{D}_{O}$ of operations on observations.

By looking at the topology of the DAG in Figure 13, we note that precedences between nodes of the two sets are all drawn from the nodes of $\mathcal{D}_{\mathcal{M}}$ to the nodes of $\mathcal{D}_{O}$. This inspires a rewrite of the DAG, shown in Figure 14, where the logical precedence of nodes of $\mathcal{D}_{\mathcal{M}}$ with respect to nodes of $\mathcal{D}_{O}$ is highlighted by the DAG's topology, and a new arc is drawn from the node M, representing the metadata results, to the `LOAD` operation relative to observations of `Input1` and `Input2`. The new precedence arc hints to a powerful optimization, i.e., the selective load of just those observation samples that contribute to the result M. Such optimization, called *meta-first*, is discussed in the next section.

[8] When a DAGs has multiple `STORE` operations, at least one of them does not depend on any stored variable; this induces a partial order of `STORE` executions.

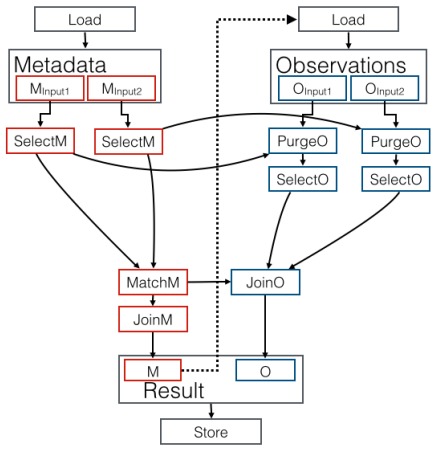

Figure 14: Metadata-first Optimization.

4.3. Metadata-first Optimization

We now introduce some notations and definitions that will be necessary to state and prove our results regarding *metadata-first optimization*.

Definition 1 (Input and output of a query)**.** *For a ScQL query Q, we write $Q(I_Q)$ to indicate the set (called* output*) of all the datasets that are written by Q to the persistent storage if I_Q (the* input*) is the set of all the datasets that are loaded by Q from the persistent storage.*

Definition 2 (Meta-equality)**.** *Two input sets I_Q and I'_Q are* meta-equal *if and only if $I_Q = \{D_1, \ldots, D_k\}$, $I'_Q = \{D'_1, \ldots, D'_k\}$ and $D_i^{\mathcal{M}} = D_i'^{\mathcal{M}}$ for $1 \leq i \leq k$.*

In other words, two input sets are meta-equal if there exists a one-to-one correspondence between the metadata of their datasets.

Definition 3 (Meta-separability)**.** *A query Q is* meta-separable *if, for every meta-equal input datasets I_Q and I'_Q, $Q(I_Q)$ and $Q(I'_Q)$ are also meta-equal.*

Intuitively, a query Q is *meta-separable* if, for every input I_Q, all metadata sets of all the datasets in $Q(I_Q)$ are independent of all of the observation sets in all of the samples of I_Q. From the DAG point of view a query is meta-separable if none of the nodes that manipulate metadata takes as input the result of any of the nodes that operate on observations. Practically, meta-separability means that all the metadata in the output $Q(I_Q)$ can be computed without even looking at the observations in I_Q. Note that the converse is not required, i.e., in a meta-separable query, the computation of the observations may still depend on the metadata in I_Q.

The example query Q_1 is meta-separable: Figure 14 shows that metadata operations precede observation operations. By construction, every query in ScQL is meta-separable except for those that contain an `EXTEND` statement. Indeed, the only intermediate representation metadata node that takes as input a set of observations is `ExtM`, which is used only within the `EXTEND` operator.

We now define the relevant notions that will allow us to efficiently compute meta-separable queries. To this end we focus on the class of queries that share the same metadata parts and isolate those sample identifiers that are potentially useful for the computation.

Definition 4 (Meta-class)**.** *Let $dag(\cdot)$ be a function that takes as input a query and returns a triplet:*

$$dag(Q) =< DAG, param_{meta}, param_{obs} >$$

where DAG is the topology of the query DAG, $param_{meta}$ a map that associates every metadata node with its parameters and $param_{obs}$ a map that associates every observation node with its parameters. For any query Q, we define the meta-class of Q as the set:

$$MC(Q) = \{Q' : \exists D, m, o, o' \quad dag(Q) =< D, m, o > \wedge \; dag(Q') =< D, m, o' >\}$$

that is the class of all the queries, including Q, with the same DAG topology and the same parameters in every metadata node.

Thanks to the previous definitions, given a query Q and an input set I_Q, we can now introduce the *meta-minimum id set* of Q with respect to I_Q.

Definition 5 (Meta-minimum id set)**.** *Consider a query Q and an input I_Q. Let $S(I_Q)$ be the set of all the IDs of all the samples in I_Q. The meta-minimum id set $S(I_Q, Q) \subseteq S(I_Q)$ of I_Q with respect to Q, is a set of identifiers such that:*

- *for every query input I'_Q meta-equal to I_Q and for every query $Q' \in MC(Q)$, if we denote $\mathcal{A}(D, S) = \{s : s \in D \wedge id(s) \in S\}$, then:*

$$Q'(I'_Q) = Q'\Big(\{\mathcal{A}(D, S(I_Q, Q)) : D \in I'_Q\}\Big)$$

- *there exists no set $S' \subseteq S(I_Q)$ for which the previous point holds and $|S'| < |S(I_Q, Q)|$.*

Given a query and an input set for which we elaborated the metadata part, the meta-minimum id set comprises only the identifiers of those samples that are necessary to correctly compute the observation part of the query output, regardless of the parameters of the observation nodes and the observations.

We now focus on meta-separable queries and describe a procedure for building the set of identifiers of the samples in the meta-minimum id set. To do this, we build the *provenance set* for each sample s, i.e., the set of identifiers of samples in I_Q that contribute to the construction of s.

Definition 6 (Provenance Set)**.** *Let Q be a ScQL query and I_Q an input. The* provenance set *$P(j, Q, I_Q)$ for sample identifier j is inductively defined as follows:*

1. *if j identifies a sample in I_Q, then $P(j, Q, I_Q) = \{j\}$;*
2. *else let `Op` be a metadata table-level operation in the DAG representation of Q and let j occur in `Op`'s output; then $P(j, Q, I_Q)$ is defined by the following rules:*

- *if* `Op` *is* `AggrM` *or* `MergeM`*: let* $\mathcal{T}$*, as in* (11)*, be the set of pairs received as input by the operator, and let* $\mathcal{G}$ *be* j*'s group, i.e.,* $\langle id(\mathcal{G}), idg(\mathcal{G})\rangle \in \mathcal{T}$ *and* $j \in idg(\mathcal{G})$*. Then,*

$$P(j, Q, I_Q) = \cup_{j' \in idg(\mathcal{G})} P(j', Q, I_Q),$$

i.e., the provenance set of each sample in the output of the operator is the union of the provenance sets of the samples involved in the grouping;

- *if* `Op` *is* `JoinM`*: let* $\mathcal{P}$*, as in* (16)*, be the set of pairs of sample identifiers involved in the join and let* $j = \nu_{j_1,j_2}$*. Then*

$$P(j, Q, I_Q) = P(j_1, Q, I_Q) \cup P(j_2, Q, I_Q),$$

i.e., the provenance set of the join sample is the union of the provenance sets of the input samples identified by j_1 *and* j_2*;*

- *no other operator modifies provenance sets (note, however, that* `SelectM`*,* `OrderM`*,* `SemiJoinM` *and* `MatchM` *may discard some samples and, thus, their corresponding provenance sets).*

The provenance set of Q *(wrt.* I_Q*) is the set*

$$P(Q, I_Q) = \bigcup_{\substack{D \in Q(I_Q) \\ s \in D}} P(id(s), Q, I_Q).$$

The construction of Definition 6 can be used to speed up the execution of the observation side of the query, by selectively loading only those samples whose identifiers are in the provenance set of Q, among those that occur in a dataset in I_Q. In other words, as claimed in Theorem 2 below, for a meta-separable query Q, the provenance set of Q correctly identifies the meta-minimum id set; therefore, once the provenance set is computed by evaluating the metadata side of Q on the entire input I_Q, the more expensive observation side can be executed on only those samples whose ids are in the meta-minimum id set.

Obviously, if two input sets I_Q and I'_Q are *meta-equal*, then for every query Q it holds that $P(Q, I_Q) = P(Q, I'_Q)$, since the provenance set is derived solely from the metadata components of the two input sets.

Theorem 2. *Let* Q *be a meta-separable query and* I_Q *an input. Then*

$$P(Q, I_Q) = S(I_Q, Q).$$

Note that the construction of Definition 4 only requires a traversal of the metadata part of the DAG. In other words, the provenance set of a meta-separable query is computed by determining the samples retained by the query, which is done by only inspecting the metadata part of the samples, while keeping track of provenance information for each processed sample.

Proof. First, note that ScQL forces a one-to-one relationship between metadata and observation sets, therefore the samples in the output of the query for which we computed the metadata set while computing the provenance set of Q are exactly the samples for which we need to compute the observation set.

Now, assume by contradiction that there exists an identifier j in $S(I_Q, Q)$ such that j is not present in the provenance set of Q. If so, one of the following situations has to have happened:

- j was not an identifier of any of the samples in I_Q, which is not possible since j is in $S(I_Q, Q)$;
- j has been removed at some point from the provenance set, which, again, is not possible since no identifier is ever removed from a provenance set (by construction);
- one of the operators that produce new identifiers (`AggrM`, `MergeM`, and `JoinM`) did not add j, but, again by construction, these operators add all the new identifiers to the corresponding provenance sets.

Contradiction.

Vice versa, assume by contradiction that there exists an identifier j such that j is in the provenance set of Q but not in $S(I_Q, Q)$. This means that at a certain point j was added to some provenance set where it was not necessary. But, the only operators that add an identifier are `AggrM`, `MergeM`, and `JoinM`, and each of them adds to the provenance set only the necessary identifiers, i.e., such that there exists an observation set that would make them part of the output and therefore in $S(I_Q, Q)$. For instance, `JoinM` adds the new identifier ν_{j_1,j_2} whenever j_1 and j_2 identify two matching samples in the input; similarly for `AggrM` and `MergeM`. Contradiction. □

Overall, our *meta-first optimization* consists in

1. identifying meta-separable queries or subqueries;
2. executing such queries or subqueries on the metadata part only, so as to construct their provenance set;
3. loading the observations of only those samples whose identifiers are in the provenance set and executing the observation part of the query on them.

Note that, implementation-wise, this optimization can be easily accommodated in any system endowed with metadata and observation management, e.g., by adopting the following principles, which are defined at the logical level and are not dependent on the data format or on the specific deploy technology:

- each provenance tuple $\langle id(s), j\rangle$ can be stored in the metadata by adding the pair $\langle$`provenance`$, j\rangle$ to $meta(s)$, with the precaution of never eliminating provenance metadata by `PROJECT` operations;
- after producing provenance sets based on the metadata side, such sets are distributed to the operations that are in charge of loading input observations, so as to selectively load their relevant samples; after that, provenance sets can be deleted.

4.3.1. Relaxing meta-separability

From the definition of meta-separability, all queries including an EXTEND statement violate the definition. Clearly, if a query mentions the new attributes that are built by an EXTEND in any operation that uses metadata predicates for filtering the samples, then such query is not separable. By construction, these operations are SELECT, JOIN, SEMIJOIN and ORDER operations with a TOP clause; we collectively call them *selective operations*. A query with EXTEND operations whose new attributes are not mentioned in selective operations is called *weakly meta-separable*. This query property can be determined by simply inspecting the query.

The execution of a weakly meta-separable query can be performed by ignoring ExtM operations when they are found during the recursive traversal of the DAG and by storing a copy of the metadata that are in input to such operations and to all subsequent operations of the DAG useful to build the metadata result. Then we proceed with the meta-first optimization of LOAD operations and the recursive descent of the observation subtree; when the BuildO operation is executed, ExtM and its subsequent operations on the DAG are recomputed, using the copies of metadata inputs, so as to deliver the correct metadata result.

EXTEND is typically used for computing statistics about observations, and, if such statistics are not further used by selective operations, then the queries remain weakly meta-separable, as only the selective operations may cause the exclusion of samples from the results.

4.3.2. Benefits of meta-first optimization

The following example presents a real-world biological computation executed on actual experiments, and shows how the efficiency of the meta-first optimization depends on the actual values that are provided at execution time. We consider the biological problem of finding the overlapping genome regions between two specific transcription factors and other regions known as "presumed enhancers". The former regions are extracted from two collections of the ENCODE repository (respectively named ENCODE_BROAD and ENCODE_NARROW), with a schema consisting of Chrom, Start and Stop. Since several samples may be present for each transcription factor, samples are merged and then joined. Presumed enhancers are produced by means of a process that extracts and extends the regions corresponding to the two antibody targets 'H3K27ac' (acetylation) and 'H3K4me1' (methylation) from the ENCODE_BROAD repository and then intersects them. The result is given by a third intersection of the two partial results by means of a join, but such intersection has to be computed for compatible cell lines; this may lead to useless region processing, as this third join, which imposes cell-line identity, is postponed to data extraction, merge, projection over suitable regions, and confirmation by the first and second joins.

This computation is expressed in ScQL by the following query pattern, where we set the second antibody target to 'BACH1', whereas we will try several different choices for the first antibody target by replacing the <AT1> placeholder with an appropriate value; L and R in the join predicates denote the left and right operands of joins.

```
LOAD ENCODE_BROAD;
LOAD ENCODE_NARROW;
broad = SELECT(antibody = <AT1>) ENCODE_BROAD;
narrow = SELECT(antibody = 'BACH1') ENCODE_NARROW;
mbroad = MERGE(cell) broad;
mnarrow = MERGE(cell) narrow;
int = JOIN(cell;
      ((L.Start <= R.Start AND L.Start <= R.Stop)
        OR
       (R.Start <= L.Start AND L.Start <= R.Stop))
        AND
       L.Chrom = R.Chrom
      ) mbroad mnarrow;

acet = SELECT(antibody = 'H3K27ac') ENCODE_BROAD;
l_acet = PROJECT(; left as (left-100),
                   right as (right+100)) acet;
met = SELECT(antibody = 'H3K4me1') ENCODE_BROAD;
l_met = PROJECT(; left as (left-100),
                  right as (right+100)) met;
en = JOIN(cell;
     ((L.Start <= R.Start AND R.Start <= L.Stop)
       OR
      (R.Start <= L.Start AND L.Start <= R.Stop))
       AND
      L.Chrom = R.Chrom
     ) l_met l_acet;

bind = JOIN(cell;
       ((L.Start <= R.Start AND R.Start <= L.Stop)
         OR
        (R.Start <= L.Start AND L.Start <= R.Stop))
         AND
        L.Chrom = R.Chrom
       ) int en;
STORE bind;
```

The join predicate used in all JOIN clauses is a common biological requirement for computing intersections of transcriptions.

We tested five different versions of the above query pattern, each corresponding to a different choice of the first antibody target. In practice, we replaced the placeholder <AT1> with values such as 'CTCF' and 'JUND', corresponding to a number of samples varying between 154 and 14, so as to cover a wide range of scenarios for tested transcription factors. The list of tested values and corresponding input samples is reported in Table 22. In all cases, the query result has 2 samples (corresponding to the cell lines K562, H1-hESC), but only a few input samples contribute to the result, and hence need to be loaded after the meta-first optimization. For instance, only 9 'CTCF' out of 154 samples are actually needed – see Table 22 for all five cases.

Experiments were performed over an Apache Spark implementation of ScQL, running on a single server equipped with a Dual Intel Xeon ES-2650 processor and 380 GB of RAM. Observation files are kept on Apache HDFS, as it is typical of many data science applications. The meta-first optimization is performed as a transformation of the load operation of the observation files. Execution times are evaluated by an implementation

of the observation operations that is essentially unchanged in the version with meta-first optimization, however operating on a much reduced input, as the only samples that contribute to the result are loaded into the Spark engine execution environment and then processed.

As for execution times, improvements are equally impressive. With the antibody target 'CTFC', it takes 858 seconds to compute the result with our prototype; however, when exploiting the meta-first optimization, execution time drops to 314 seconds (2.73x improvement). The smallest improvement corresponds to antibody target 'ELAVL1', in which execution time drops from 298 seconds to 246 seconds with the meta-first optimization, thus still obtaining a non-negligible 1.22x improvement. Note that execution time improvements exceed 2x in 4 cases out of 5. Table 22 reports all tested cases.

Note that the metadata-first optimization relies both on a *compile-time* modification of the query plan and on a *runtime* computation of the provenance set which, despite its logical nature, cannot be inferred from the text of the query. It is an optimization because the cost of computing operations on metadata is negligible when compared to the cost of loading observations and of computing operations on them. Also, note that it is possible to build arbitrarily large improvements, e.g., by using in a biological query transcription factors that have no cell in common, or by building artificial data whose meta-join condition fails. Having ascertained the benefits of the meta-first optimization both in a practical, real-world case and with theoretical considerations, we think that showing other experiments for corroborating this point would be of little use. Instead, in the remainder of the section we shall discuss further opportunities for optimization. Further scenarios of applicability of the meta-first optimizations are discussed in Section 5.

4.4. Other Logical Optimizations

Standard logical optimizations apply to the computation of a DAG by considering table-level operators. In particular, it is relevant to focus on meta-separable queries and to consider the sub-DAG that builds the observations of the result, for which classical relational optimization techniques are most significant (note that observations have a schema, regardless of how they are stored).

In addition, the operator tree is a classical tree of relational operations, and those operations that reduce the sizes of operands can be anticipated thanks to operation commutativity; in particular, they can be pushed to the `LOAD` operations. `SelectO` commutes with no changes with all the operations over observations; `ProjectM` commutes with `SelectM`, `JoinM`, `OrderM` and `AggrM` after extending the list of projection attributes with the attributes mentioned by these operations.

5. Applicability of the approach

We already presented comprehensive examples from the domain of genomics in the previous section. We now present examples from two more domains: software management and social analytics.

5.1. Commits on Github

Among several big data sources, Google is publishing Github commits[9]. GitHub is home to the largest community of open source developers in the world, with over 12 million people contributing since 2008; the 3TB+ dataset comprises a full snapshot of the content of more than 2.8 million open source GitHub repositories, including more than 145 million unique commits, each with an observation consisting of a very large record, whose schema includes the filename, id and path of the committed file and the difference between the current and previous version. Commits can be searched by project, date, and author, and we can consider these as part of the metadata of each commit observation. As a general observation, when data sources support an API for retrieval, it is possible to load them selectively by making use of their API, by considering the API input parameters as metadata.

We then consider an Apache dataset, where each sample is a project; there is one observation for each version (with date and note), whereas the metadata include generic aspects about the Apache project, including the project's name, category, and license.

We then consider a query extracting from the Github dataset the authors of commits of Apache projects of the `cloud` category. This piece of information is not directly available on the Github repository, thus a semi-join with the Apache project is required. The corresponding ScQL query is:

```
LOAD GITHUB_COMMITS;
A = EXTEND(modified as count(*)) GITHUB_COMMITS;
B = GROUP(author, total as SUM(modified)) A;
C = SELECT(category = 'cloud') APACHE_PROJ;
result = SEMIJOIN(project; C) B;
STORE result;
```

This program can be improved according to several optimizations:

- Standard algebraic optimization can be used for anticipating the semi-join before the extend operation.
- Metadata-first optimization applies, because the attribute `modified`, which is built by the `EXTEND` operation, is not used in any subsequent selective operation; thus, the query is weakly separable.

By effect of meta-first, instead of loading 176,157,745 observations regarding commits, it is possible to load just 22,233 commits relative to apache cloud projects (i.e., just 0.01% of the commits); all apache projects are 731,698, thus the meta-first optimization for a query with no selection on `cloud` would amount to loading only 0.41% of the overall commits.

Note that the `ORDER` with `TOP` operation can be applied to the result; e.g., it is possible to select the ten most prolific contributors (by number of commits) of the Apache projects with category set to `cloud`, as follows:

[9] https://cloud.google.com/bigquery/public-data/github

	Without Meta-First		With Meta-First		Gain factor	
AT1	Loaded Samples	Execution time	Loaded Samples	Execution time	Samples	Time
CTCF	154	858s	9	314s	17.11	2.73
H3K4me3	150	827s	14	324s	10.71	2.55
POLR2A	58	320s	9	138s	6.44	2.31
ELAVL1	15	298s	6	246s	2.5	1.21
JUND	14	78s	2	32s	7	2.4

Table 22: Benefits of the Meta-First Optimization on the number of loaded samples and on execution time.

```
top = ORDER(total DESC, TOP 10) result;
```

Top contributors are commit authors, and their identity is found by looking at the metadata attribute of the resulting dataset. Note that, although the overall query is not weakly meta-separable (because of the `TOP` construct), breaking it in two parts (first the weakly meta-separable query for computing `result`, then the query for `top`), as we did here, allowed us to benefit from the gain of the meta-first optimization anyway.

5.2. Social analytics

Consider the posts about the world's most important fashion weeks (Milano, London, Paris, New York), collected from two social sources: Instagram and Twitter. We consider the problem of identifying the fashion brands which were most represented, on both social sources, at anyone of these event, and of identifying their posts. We assume that the hashtags of the most popular brands are known to experts; then, in a data preparation phase, posts can be collected using suitable APIs. Specifically,

- Twitter has a powerful API where it is possible to compose queries over brand/week pairs. We build a dataset called `TWEETBYBW` by setting a brand/week hashtag pair as mandatory in each call to the API, thus producing one sample for each pair of event and brand, with metadata attributes including the *event* (valued as one of `#MLW, #LFW, #PFW, #NYFW`) and the *brand* (e.g, `#GUCCI`).
- Instagram's API does not allow multiple mandatory terms in requests. In this case, we build a dataset called `INSTABYW` by assembling within a dataset the results of simple requests based on the four events (with one metadata attribute `event`) and `INSTABYB` using simple requests based on the brand (with one metadata attribute *brand*). Given that Instagram posts have an identifier, a join on the common identifier of `INSTABYW` and `INSTABYB` returns event/brand pairs with the intersection of the observations and with metadata *brand, event* as in Twitter, as follows[10]:

```
INSTABYBW = JOIN (id;) INSTABYB, INSTABYW;
```

Then, a ScQL query must extend both `TWEETBYWB` and `INSTABYWB` with the counters of posts, join the two collections on the metadata `Brand` and `Week`, compute a weighted sum of the counters, and extract the top-k elements; the result is the set of observations combining posts from Instagram and Twitter, whose metadata contain the week/brand pairs associated with the highest weighed counts. The ScQL Query is composed by two parts:

```
LOAD INSTABYBW;
LOAD TWEETSBYBW;
A = EXTEND(Count_I AS Count(*)) INSTABYBW;
B = EXTEND(Count_T AS Count(*)) TWEETSBYBW;
result = JOIN(Brand,Week) A,B;
STORE result;
```

Note that this query is weakly separable, as metadata are computed by `EXTEND` operations are not used by any selective operation. Thus, it is possible to perform the meta-first optimization. In practice, the optimization has no effect if the brand list includes few most popular brands, but can be significant if the brand list includes several thousand small brands. We then add to the query the following part:

```
C = PROJECT(Weight=Count_T + 10 * Count_I) result;
top = ORDER(Week; Weight, TOP 10) C;
STORE top;
```

This second part, which is not meta-separable, extracts the top 10 weighted brands for every fashion week.

6. Related Work

The importance of metadata in big data management is now widely recognized, and the appearance of many systems and tools accommodating metadata as first-class citizens (e.g., [8]) testifies to a general consensus on their crucial role in scientific workflows systems to achieve important functionalities, such as workflow and service discovery, composition and provenance browsing, to name a few. In 2006, Geerts et al. proposed MONDRIAN [9], an annotation oriented data management system. Many aspects of the concept of annotation proposed by the authors are similar to ScQL metadata; both allow one to track the provenance of the observation data and to enrich those with additional information. However, strong differences exist and motivate the introduction of ScQL. First of all MONDRIAN annotations are fine grained (an annotation can refer to single tuples or even to a subset of the fields of a tuple); conversely, in the ScQL model every metadata pair refers to all the observation in its sample. This allows us to define set-oriented operations on

[10] To avoid the double extraction of Instagram posts, one can collect just `INSTABYW`, and then filter the posts specific to each brand by adding at the same time their event metadata, with a custom data preparation.

metadata, which would not be feasible in MONDRIAN, such as: grouping and ordering samples according to metadata values, moving information from metadata to observation and vice versa and joining observations on metadata.

Xiao et al. in 2014 proposed an annotation-oriented database system named InsightNotes [10]. Compared to ScQL, it provides a data model for annotation (metadata), that culminates in the notion of *Annotation Summaries*. The main concern of the paper is very different from ours, as it focuses on knowledge extraction based on such summaries through machine learning techniques.

Recent works in the context of metadata focused more on representation and collaboration issues than on query processing aspects [11, 12]. In our approach, metadata are additionally used in order to reconstruct the provenance of the different data samples involved in a query. Provenance, too, is a well understood topic in many data contexts, including scientific databases, where it is perceived as a crucial component for ensuring reproducibility of scientific analyses and processes [13, 14, 15] as well as validation of experiments [16]. Several joint efforts have been targeting a better and more interoperable use of provenance, such as the Open Provenance model [17].

A few other works in the context of scientific data have extended traditional relational query languages to include metadata capabilities of some kind. Among these, we mention the context of multi-dimensional grids or "arrays", which have received a long-standing attention [18, 19, 20]. Arrays in practice are usually accompanied by metadata, which need a fruitful integration with the processing paradigms of the DBMS, and often require dedicated systems and models; among these, rasdaman [21], SciDB [22], SciQL [23, 24], and ASQL [25]. Similarly to the ASQL model, we assume that scientific databases are ornamented with metadata, and aim to integrate this kind of information during query processing.

Unlike previous works, our use of metadata gives rise to a new notion of provenance, moving from object provenance to *sample provenance*, whose goal is to provide collective properties of the results built during query processing, based on the selected input samples and operations. In this respect, our notion of sample provenance differs from the notion of record-based provenance discussed in seminal work in the database community [26, 27, 28] (see [29] for a survey on database provenance). Such works focus on tracing the provenance of each individual record forming the result, typically in the context of selective queries producing very specific results. But in the context of data science, with millions of records representing physical phenomena and thousands of files collecting them, and with queries typically producing big datasets to be used for subsequent data analysis, the most interesting notion of provenance is at the file level, as is done in sample provenance – also because each record is not associated with specific metadata, whereas files are. Sample provenance and record-based provenance are therefore not comparable, as they address distinct problems.

Some systems, such as Perm [30] and Orchestra [31], support fine-grained provenance computations by rewriting SQL queries. Although these systems support a more powerful notion of provenance, their focus is different. In particular, Orchestra aims to enable data sharing across locally controlled peers related by a network of schema mappings. The query semantics depends then on the mappings, and the result tuples are associated with probabilities (e.g., ranging between "certain" results, when their derivation occurs according to all possible sources and mappings, and "uncertain results", whose trust scores depend on the trustworthiness of the sources). In the data science contexts we consider, all sources are trusted.

One of the key features of ScQL is that it leverages the co-occurrence of metadata and observations to provide superior optimization opportunities, based on the meta-first approach. The literature regarding logical query optimization is immense, starting from [32] and subsequent works. Most works exploit schema knowledge or integrity constraints [33, 34] to perform compile time query optimization [35, 36], both for traditional database systems and more recent systems. In stream databases, for example, dynamic metadata have been used for optimization purposes focusing on single operations such as joins [37, 38] as well as on compile-time detection of "unsafe" queries [39]. The work [40] uses semantic query optimization utilizing dynamic metadata at runtime to find better query plans than those selected at compile time. Although sharing the objective of improving query processing, [40] attempts to find a different (better) query plan based also on metadata, while our approach single-handedly improves the selectivity of the operators involved in a query and potentially discarding the loading of many samples, without changing the logical query plan.

In [41] provenance is used to improve system performance. Despite the overall approach has some similarities with metadata-first optimization (i.e., both mechanisms rely on narrowing the initial input) the two methods have strong differences. First of all the concept of provenance adopted in [41] is extremely fine-grained (provenance predicates are computed for every element with respect to every transformation in the workflow that produces it) while in our case the provenance is at the level of samples. Moreover, [41] uses provenance to efficiently perform the selective refresh of single elements, in contrast to our approach where we leverage sample provenance to optimize the evaluation of the whole query.

GMQL [3, 5] is a specialization of ScQL, with emphasis on biological operations for region management; its application to a variety of biological problems dealing with heterogeneous datasets is discussed in [3], whereas its efficient implementation is discussed in [5]. This paper and [5] are complementary, as [5] is focused on domain-specific, region management operations (called `Genometric Join`, `Cover` and `Map`) and does not discuss metadata management, whereas this paper is focused on domain-independent relational operations and on metadata management. All domain-specific operations are meta-separable, hence the meta-first optimization is applicable to them, yielding to significant computational savings; however, proving this claim requires the lengthy descriptions of domain-specific GMQL operations and is outside of the scope of this paper.

7. Conclusions

ScQL is a domain-independent, algebraic relational language giving the same relevance to metadata and to observations. It is applicable whenever a dataset consists of several homogeneous samples, each associated with a given experimental condition; metadata describe the experimental conditions using a simple format, consisting of attribute-value pairs. We have provided the formal specification of ScQL, shown that it can be applied to several domains, introduced meta-separable queries and proved that the meta-first optimization can be applied to them. ScQL was inspired by GMQL, a language for data-centric genomic computing that is fully implemented on the Spark and Flink cloud-based database engines. However, ScQL generalizes GMQL, and has a much broader applicability, as it can be used with arbitrary scientific domains.

References

[1] E. P. Consortium, et al., An integrated encyclopedia of dna elements in the human genome, Nature 489 (2012) 57–74.

[2] J. N. Weinstein, E. A. Collisson, G. B. Mills, K. R. M. Shaw, B. A. Ozenberger, K. Ellrott, I. Shmulevich, C. Sander, J. M. Stuart, C. G. A. R. Network, et al., The cancer genome atlas pan-cancer analysis project, Nature genetics 45 (2013) 1113–1120.

[3] M. Masseroli, P. Pinoli, F. Venco, A. Kaitoua, V. Jalili, F. Palluzzi, H. Muller, S. Ceri, Genometric query language: a novel approach to large-scale genomic data management, Bioinformatics 31 (2015) 1881–1888.

[4] T. Hey, S. Tansley, K. M. Tolle, et al., The fourth paradigm: data-intensive scientific discovery, volume 1, Microsoft research Redmond, WA, 2009.

[5] A. Kaitoua, P. Pinoli, M. Bertoni, S. Ceri, Framework for supporting genomic operations, IEEE Transactions on Computers (2016).

[6] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig latin: a not-so-foreign language for data processing, in: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, ACM, pp. 1099–1110.

[7] E. F. Codd, Relational completeness of data base sublanguages, IBM Research Report RJ987 (1972).

[8] K. Belhajjame, K. Wolstencroft, Ó. Corcho, T. Oinn, F. Tanoh, A. R. Williams, C. A. Goble, Metadata management in the taverna workflow system, in: 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2008), 19-22 May 2008, Lyon, France, pp. 651–656.

[9] F. Geerts, A. Kementsietsidis, D. Milano, Mondrian: Annotating and querying databases through colors and blocks.

[10] D. Xiao, M. Y. Eltabakh, Insightnotes: summary-based annotation management in relational databases, in: Proceedings of the 2014 ACM SIGMOD international conference on Management of data, ACM, pp. 661–672.

[11] E. Kandogan, M. Roth, P. M. Schwarz, J. Hui, I. Terrizzano, C. Christodoulakis, R. J. Miller, Labbook: Metadata-driven social collaborative data analysis, in: 2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015, pp. 431–440.

[12] K. P. Smith, L. J. Seligman, A. Rosenthal, C. Kurcz, M. Greer, C. Macheret, M. Sexton, A. Eckstein, "big metadata": The need for principled metadata management in big data ecosystems, in: Proceedings of the Third Workshop on Data analytics in the Cloud, DanaC 2014, June 22, 2014, Snowbird, Utah, USA, In conjunction with ACM SIGMOD/PODS Conference, pp. 13:1–13:4.

[13] Y. Simmhan, B. Plale, D. Gannon, A survey of data provenance in e-science, SIGMOD Record 34 (2005) 31–36.

[14] S. B. Davidson, J. Freire, Provenance and scientific workflows: challenges and opportunities, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008, pp. 1345–1350.

[15] Y. Gil, E. Deelman, M. H. Ellisman, T. Fahringer, G. C. Fox, D. Gannon, C. A. Goble, M. Livny, L. Moreau, J. Myers, Examining the challenges of scientific workflows, IEEE Computer 40 (2007) 24–32.

[16] S. Miles, S. C. Wong, W. Fang, P. T. Groth, K. Zauner, L. Moreau, Provenance-based validation of e-science experiments, J. Web Sem. 5 (2007) 28–38.

[17] N. Kwasnikowska, L. Moreau, J. V. den Bussche, A formal account of the open provenance model, TWEB 9 (2015) 10:1–10:44.

[18] P. Baumann, Management of multidimensional discrete data, VLDB J. 3 (1994) 401–444.

[19] L. Libkin, R. Machlin, L. Wong, A query language for multidimensional arrays: Design, implementation, and optimization techniques, in: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996., pp. 228–239.

[20] A. P. Marathe, K. Salem, Query processing techniques for arrays, VLDB J. 11 (2002) 68–91.

[21] A. Andrejev, T. Risch, Scientific SPARQL: semantic web queries over scientific data, in: Workshops Proceedings of the IEEE 28th International Conference on Data Engineering, ICDE 2012, Arlington, VA, USA, April 1-5, 2012, pp. 5–10.

[22] M. Stonebraker, P. Brown, A. Poliakov, S. Raman, The architecture of scidb, in: Scientific and Statistical Database Management - 23rd International Conference, SSDBM 2011, Portland, OR, USA, July 20-22, 2011. Proceedings, pp. 1–16.

[23] Y. Zhang, M. L. Kersten, S. Manegold, Sciql: array data processing inside an RDBMS, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013, pp. 1049–1052.

[24] Y. Zhang, M. L. Kersten, M. Ivanova, N. Nes, Sciql: bridging the gap between science and relational DBMS, in: 15th International Database Engineering and Applications Symposium (IDEAS 2011), September 21 - 27, 2011, Lisbon, Portugal, pp. 124–133.

[25] D. Misev, P. Baumann, Homogenizing data and metadata retrieval in scientific applications, in: Proceedings of the ACM Eighteenth International Workshop on Data Warehousing and OLAP, DOLAP 2015, Melbourne, VIC, Australia, October 19-23, 2015, pp. 25–34.

[26] Y. Cui, J. Widom, Lineage tracing in a data warehousing system, in: Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000, pp. 683–684.

[27] P. Buneman, S. Khanna, W. C. Tan, Why and where: A characterization of data provenance, in: Database Theory - ICDT 2001, 8th International Conference, London, UK, January 4-6, 2001, Proceedings., pp. 316–330.

[28] T. J. Green, G. Karvounarakis, V. Tannen, Provenance semirings, in: Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China, pp. 31–40.

[29] J. Cheney, L. Chiticariu, W. C. Tan, Provenance in databases: Why, how, and where, Foundations and Trends in Databases 1 (2009) 379–474.

[30] B. Glavic, G. Alonso, Perm: Processing provenance and data on the same data model through query rewriting, in: Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China, pp. 174–185.

[31] T. J. Green, G. Karvounarakis, Z. G. Ives, V. Tannen, Update exchange with mappings and provenance, in: Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007, pp. 675–686.

[32] A. K. Chandra, P. M. Merlin, Optimal implementation of conjunctive queries in relational data bases, in: Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA, pp. 77–90.

[33] R. Hull, R. King, Semantic database modeling: Survey, applications, and research issues, ACM Comput. Surv. 19 (1987) 201–260.

[34] D. Tsichritzis, F. H. Lochovski, Data Models, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[35] P. M. Fischer, K. S. Esmaili, R. J. Miller, Stream schema: providing and exploiting static metadata for data stream processing, in: EDBT 2010, 13th International Conference on Extending Database Technology, Lausanne, Switzerland, March 22-26, 2010, Proceedings, pp. 207–218.

[36] J. J. King, QUIST: A system for semantic query optimization in relational databases, in: Very Large Data Bases, 7th International Confer-

ence, September 9-11, 1981, Cannes, France, Proceedings, pp. 510–517.

[37] L. Ding, N. K. Mehta, E. A. Rundensteiner, G. T. Heineman, Joining punctuated streams, in: Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings, pp. 587–604.

[38] L. Ding, E. A. Rundensteiner, Evaluating window joins over punctuated streams, in: Proceedings of the 2004 ACM CIKM International Conference on Information and Knowledge Management, Washington, DC, USA, November 8-13, 2004, pp. 98–107.

[39] H. Li, S. Chen, J. Tatemura, D. Agrawal, K. S. Candan, W. Hsiung, Safety guarantee of continuous join queries over punctuated data streams, in: Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006, pp. 19–30.

[40] L. Ding, K. Works, E. A. Rundensteiner, Semantic stream query optimization exploiting dynamic metadata, in: Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany, pp. 111–122.

[41] R. Ikeda, S. Salihoglu, J. Widom, Provenance-based refresh in data-oriented workflows, in: Proceedings of the 20th ACM international conference on Information and knowledge management, ACM, pp. 1659–1668.