The PhotoSpread Query Language

Sean Kandel, Andreas Paepcke, Martin Theobald
and Hector Garcia-Molina
{skandel,paepcke,theobald,hector}@cs.stanford.edu

Stanford University

September 6, 2007

Abstract

This document defines the data model as well as the syntax and semantics of the formula language employed by *PhotoSpread*. It is inspired by Excel with specialized and enriched functionality for managing and tagging large photo collections in a spreadsheet. PhotoSpread allows for capturing, storing, arranging, manipulating, and querying arbitrary tagged photo objects with an intuitive and easy-to-learn, yet expressive formula language.

1 Main Features at a Glance

- A spreadsheet is an ordered set of cells.
- Cells are always associated with a container and a formula.
- A container stores an unordered set of objects (with layout information stored through tags).
- A formula refers to (i.e., selects) an unordered set of objects which are the ones to be displayed in the cell.
- An object can represent a photo or the value of a simple data type, optionally annotated with an unordered set of user-defined tags.
- Tags are attribute-value pairs, with various ways of representing their logical connections.
- Aggregations over sets again yield another unordered result set of objects (possibly with empty tag sets).
- Thus all operators are (i.e., the entire language is) closed and complete.
- Operators are by default duplicate eliminating (over sets of objects).

2 Data Model

2.1 Spreadsheets

A *spreadsheet* S is a two-dimensional table of cells. Each spreadsheet has a unique name which may confirm either to its filename on disk or to a schema name in a relational database. Each *cell* $ij \in S$ in a spreadsheet is referred to by its *column-id* i, denoted by capital letters in lexical order, and its *row-id* j, denoted by integers in numerical order. The composed id ij is called a *cell reference*.

Examples of valid cell references:

```
A1
D24
CB89
```

Furthermore, row and column identifiers in cell references can be fixed by a '`$`' in order to make the cell reference immutable to copy and move operations. For example:

```
$CB$89
```

2.2 Cells, Containers & Formulas

Each cell ij is associated with an unordered set of objects, the so-called container C_{ij}, which is initially empty, i.e., $C_{ij} = \emptyset$, and a formula F_{ij} (possibly having references to other cells and their containers or formulas), which initially references only the cell that it is associated with, i.e., $F_{ij} = ij$.

A *container* C_{ij} stores an unordered set of objects $\{O_{ij,1}, \ldots, O_{ij,k}\}$. The spreadsheet always displays the set of objects confirming to the evaluated formula expression in each cell (see Section 3 for valid formula expressions and their evaluation).

Every cell must have a formula associated with it. If the user deletes or drops a formula in cell ij, the formula is automatically reset back to its own cell reference, i.e., $F_{ij} = ij$, and the content of container C_{ij} is displayed in cell ij.

2.3 Cell Ranges

Cell ranges of the form $ij : nm$ refer to an unordered subset of the spreadsheets' cell references delimited by the top-left and bottom-right borders of cell references. The column-id i in the left cell reference in a cell range must be of lower or equal lexical order than the column-id n in the right cell reference, and the row-id j in the left cell reference must be of lower or equal numerical order than the row-id m of the right cell reference. That is:

$$ij : nm = \bigcup_{i \leq a \leq n, j \leq b \leq m} ab$$

For the special case when both column ids and both row ids are equal, the cell range evaluates to the empty set, i.e., $ij : ij = \emptyset$.

Examples of valid cell ranges:

```
A1:C3
B1:BC1
A1:A1
```

2.4 Objects

An *object* O_k can either represent the value of a basic data type (*string*, *integer*, *decimal number*, or *time* and *date* entry) that is directly entered into the spreadsheet by a user, or it can represent a photo object that is directly captured from an external source such as a URL, a file, or even from a pointer to a social network resource like Flickr[1].

This notion of “objects” is intended to provide a uniform means of storing and capturing different entities along with a brief, structured description, the so-called *tags*. The process of adding this kind of structured information to an object is called *tagging*. The list of supported entities could easily be extended to different media such as audio and video, which would not affect the data model but merely the way cell contents are displayed and presented to the user.

2.5 Tags

Each object is an unordered set of attribute-value pairs, or *tags* t_l. All objects stored in a container have a unique identifier denoted by the reserved attribute name `id` with an integer value, as well as a reserved attribute with the name `value` whose value (i.e., a link to a photo, or a basic data type) represents the actual value displayed in the cell of the spreadsheet. Any object may furthermore be tagged by an unordered set of arbitrary, user-defined attribute-value pairs, with names other than `id` or `value`.

2.5.1 Logical Structure

An *attribute name* a is always a string, whereas an *attribute value* v is itself of a basic data type (string, integer, etc.), or it is a set of basic data types that each represents mutually exclusive alternatives of the attribute's value. Thus, any object may have multiple attributes with the same name but different values, which are evaluated in a conjunctive manner; and, furthermore, each attribute can be assigned a set of distinct

[1]http://www.flickr.com/

values (separated by a semicolon in the spreadsheet editor). In the latter case, multiple values of the same attribute are considered to be mutually exclusive (see below for details on how expressions over these value sets are evaluated). In the following we will overload the notion of an attribute value to be either a set or a single value, and we explicitly refer the set representation only when needed.

Example representation of an object's tag set (this might be extended/replaced by a screenshot):

```
id=64378
value=http://...
species=squirrel
species=fox;cat
location=south ridge
temperature=54.6
time=4:30
```

In the above example, the two tags with the name `species` would denote that this photo has a squirrel *and either* a fox *or* a cat on it, but not a fox and a cat at the same time. See Section 3.2 for the definition of the exact semantics of filter expressions over these tags structures.

2.5.2 Additional Metadata

In addition to the manually assigned tags, the system may automatically derive and store more metadata about the object's *creation date*, *positioning* and *layout* in the cell, as well as other *image metadata*, for example derived from the metadata provided by the JPEG file, camera or other sensors, and may add this information directly into the object's set of tags.

3 Formula Language

Formulas F_{ij} may not only refer to the cell ij that they are directly stored in, but may be composed into expressions over multiple cell references or even other formulas. Thus, a *formula expression* is a composition of *cell references* or *cell ranges*, *set operators*, *filter predicates*, *value selections*, and object-type-specific *aggregations*. Each of these components is optional. Valid compositions of formula expressions are defined by the grammar provided in Section 4.

A formula expression may be either a *base formula* or a *derived formula*. A base formula only contains references to cells ij whose formula F_{ij} in turn refers to the cell itself (i.e., $F_{ij} = ij$), whereas a formula is called a derived formula if it contains at least one cell reference to a cell with a formula that references another cell or formula (i.e., $F_{ij} \neq ij$). The evaluation of a direct cell reference in a formula $F_{ij} = ij$ always returns the set of objects stored in the container C_{ij}.

Cell references in formulas must be *acyclic*, i.e., any two sets of cell references occurring in the derivation of a formula must be disjoint. Thus, the cell references of each derived formula always form a directed acyclic graph (DAG).

Any valid formula expression starts either with a direct *cell reference*, a *cell range*, a *set operator*, or an *aggregation*. The evaluation of a formula always returns a set of objects, thus our formula language is *closed* under the below operators. In the following, we define the semantics of each of these syntactical constructs of the language. We will also use the terms 'containers' and 'formulas' as shorthand for actual cell references of the respective type.

3.1 Set Operators

Set operators follow the usual set semantics and comprise `union`, `intersect`, and `minus`. Set operators are by default duplicate-eliminating. They are defined to take both containers or other formulas as arguments, and they can operate over arbitrary sets of objects (which may contain or represent both basic data types and photos).

Arguments of set operators can be of variable length and are evaluated from left to right.

Examples of formulas with set operators:

```
union(C1,C2,D3,C4)
intersect(C1,C2)
minus(C1,C2,C3)
```

3.1.1 Transformation Rules for Set Operators

Basic transformation rules:

```
union(C1) ≡ C1
intersect(C1) ≡ C1
minus(C1) ≡ C1

union(C1,...,Cn)
 ≡ union(union(union(C1, C2), C3),...,Cn)

intersect(C1,...,Cn)
 ≡ intersect(intersect(intersect(C1, C2), C3),...,Cn)

minus(C1,...,Cn)
 ≡ minus(minus(minus(C1, C2), C3),...,Cn)
```

Similar transformation rules hold for cell ranges:

`union(A1:B2,C3)` $\equiv$ `union(A1,A2,B1,B2,C3)`

But:

`intersect(A1:B2,C3)` $\equiv$ `intersect(union(A1,A2,B1,B2),C3)`

`minus(A1:B2,C3)` $\equiv$ `minus(union(A1,A2,B1,B2),C3)`

All set operators are *closed* under this semantics.

NOTE: Furthermore, this representation is also *complete*, i.e., we can represent any finite set of objects in a spreadsheet and express their relationships within formulas following the usual set semantics.

3.2 Filter Predicates

Cell references, cell ranges, and set operators in a formula expression may be constrained by *filter predicates.* Filter predicates operate on an object's set of tags and select those objects in a container referenced by a formula that match the filter expression $filterexp$ (see below for what a filter expression and a respective match is).

Examples of cell references, cell ranges, and set operators with filter predicates:

`C1[`$filterexpr$`]`
`C1:DF45[`$filterexpr$`]`
`union(C1,C4[`$filterexpr_1$`])[`$filterexpr_2$`]`

In each of the above examples, only those objects in the referenced cells are selected for which *filterexpr* evaluates to `TRUE`.

3.3 Filter Expressions

A *filter expression* is a boolean expression, including parentheses, logical `and`, `or`, and `not`, over attribute-value pairs. We will use '`&`', '`|`', and '`!`' as shorthand in formulas.

Attribute value pairs are matched either based on an exact match ('`=`') between the attribute name of an object and its value, or based on a data-type-specific range ('`<`', '`<=`', '`>=`', or '`>`') between the matching attribute and its value, each specified by a respective attribute-value condition and their logical connection in the filter expression.

For example, the formula

`C1[species=fox]`

selects all objects referred to by the formula associated with cell `C1` that have at least one tag with attribute name '`species`' whose value equals '`fox`'.

3.3.1 Transformation Rules for Filter Expressions

The usual axiomatic conversion rules for Boolean Algebra hold for filter expressions, including associativity, commutativity, absorption, distributivity, and complement, as well as De Morgan's law.

Example of a formula with a filter predicate and a valid boolean conversion of its filter expression (using De Morgan's law):

```
C1[species=fox & !(time=12:00|temperature=0.6)]
 ≡
C1[species=fox & time!=12:00 & temperature!=0.6]
```

3.3.2 Wildcards

The specification of an attribute name in a filter expressions may be skipped by a *wildcard* '`*`', which selects any object referenced by the formula, where *any* attribute name matches the given value. Conversely, values of attributes may be omitted, or may denoted by a wildcard '`*`' in the filter expression as well.

Examples of valid formulas with wildcards in their filter predicates:

```
C1[*=fox]
C2[species=*]
```

Further valid abbreviations (and their transformations) are:

```
C2[species] ≡ C2[species=*]
C2[*] ≡ C2[*=*] ≡ C2
```

3.3.3 Semantics of Filter Expressions

Given a set of objects $O = \{O_{ij,1}, \ldots, O_{ij,k}\}$ with tags $t_{l,p}$ consisting of attribute-value pairs with attribute names $a_{l,p}$ and attribute values $v_{l,p}$ for $l = 1, \ldots, k$, a filter expression of the form $a = v$ selects objects $o_p \in O$ such that:

$$\{o_p \in O \mid \exists\ t_{p,r} \in o_p \ with \ a_{p,r} = a \ \wedge \ v_{p,r} = v\}$$

Conjunctions in filter expressions of the form $a_1 = v_2$ `and` $a_2 = v_2$ are modeled as:

$$\left\{ o_p \in O \mid \exists\, t_{p,r_1}, t_{p,r_2} \in o_p \ \mathit{with} \ \left| \begin{array}{l} a_{p,r_1} = a_1 \ \wedge \ v_{p,r_1} = v_1 \ \mathit{and} \\ a_{p,r_2} = a_2 \ \wedge \ v_{p,r_2} = v_2 \end{array} \right. \right\}$$

(With analogous formulations for *disjunction* and *negation.*)

For the special case where multiple values are assigned to a single tag (see Section 2), *mutual exclusiveness* of alternative attribute values $v_{l,p,1}, \dots,$ $v_{l,p,q}$ for a filter expression of the form $a = v_1$ **and** $a = v_2$ is modeled as:

$$\left\{ o_p \in O \mid \ \exists\, t_{p,r_1}, t_{p,r_2} \in o_p \ \mathit{with} \ \begin{array}{l} a_{p,r_1} = a \ \wedge \ v_{p,r_1} = v_1 \ \mathit{and} \\ a_{p,r_2} = a \ \wedge \ v_{p,r_2} = v_2 \ \mathit{and} \\ r_1 \neq r_2 \end{array} \right\}$$

That is, if cell `C1` contains the following two example photo objects (including the one from Section 2)

$O_1 =$
```
{id=1,
 value=http://...,
 species=squirrel,
 species=fox;cat,
 location=south ridge}
```
and

$O_2 =$
```
{id=2,
 value=http://...,
 species=squirrel,
 species=fox,
 species=cat}
```

then some example formulas are evaluated as follows:

`C1[species=squirrel]` $= \{O_1, O_2\}$
`C1[species=fox]` $= \{O_1, O_2\}$
`C1[species=squirrel & species=fox]` $= \{O_1, O_2\}$
`C1[species=squirrel & species=cat]` $= \{O_1, O_2\}$
`C1[species=squirrel & species=fox & species=cat]` $= \{O_2\}$

`C1[species=fox | species=cat]` $= \{O_1, O_2\}$
`C1[species!=fox | species!=cat]` $= \{O_1, O_2\}$
`C1[species!=fox & species!=cat]` $= \emptyset$
`C1[species=fox & species!=cat]` $= \{O_1\}$

`C1[(species=fox & species!=cat)|(species!=fox & species=cat)]`$=$ $\{O_1\}$

3.4 Value Selections

Attribute values of objects may be selected by a *value selection* expression in order to form a new set of objects, which are initially all containing an empty set of tags. The return type of a value selection is again a set of objects representing the values of the corresponding data type of the selected object's attributes. Note that not all values of an attribute referred to by a value selection necessarily need to correspond to the same data type (neither among value selections over a single referenced object nor across objects).

Example formulas with value selections:

```
C1[species=fox].location
C1[species=fox].*
```

Value selections are by default duplicate-eliminating. As shorthand for assigning values to tags in a formula, we allow direct cell references in right-hand side of the assignment:

`C1[species=A1]` $\equiv$ C1[species=A1.species]

3.5 Aggregations

Aggregations over attribute values follow the classic, data-type-specific semantics and may be overloaded for objects representing different data types. They include `min`, `max`, `sum`, `count`, and `avg`. Aggregations over a set of different object types are generally not defined (except for `count`) and return an empty result set.

Example formulas with aggregations:

```
count(C1.species)
count(count(C1.species))
```

[**NOTE:** Filter predicates and aggregations together could now easily be extended to the full expressiveness of SQL, including analogous constructs for the `GROUP BY` and `HAVING` clauses.]

4 Grammar for Formulas

The composition of formula expressions must adhere to the following context-free grammar:

```
formula     => cellref
             | cellrange
             | setoperator
             | aggregation
setoperator => setoprname(setargs)
             | setoprname(setargs).attrname
setoprname  => union
             | intersect
             | minus
aggregation => aggrname(formula)
             | aggrname(formula).attrname
aggrname    => min
             | max
             | sum
```

```
            | count
            | avg
setargs     => formula
            | formula,formula
cellref     => filterpred
            | filterpred.attrname
cellrange   => filterpred:filterpred
            | filterpred:filterpred.attrname
filterpred  => cellid
            | cellid[filterexp]
filterexp   => ( filterexp & filterexp )
            | ( filterexp || filterexp )
            | !(filterexp)
            | attrname = attrvalue
            | attrname != attrvalue
            | attrname > attrvalue
            | attrname >= attrvalue
            | attrname < attrvalue
            | attrname <= attrvalue
cellid      => {$} {a-z}+ {$} {0-9}+
attrname    => {a-z, 0-9}+
            | *
attrvalue   => {a-z, 0-9}+
            | {0-9}+
            | *
```

5 Optimization of Formula Expressions

5.1 Basic Axioms

In addition to the usual axioms of set theory, for *base formulas* (see Section 2) the following axioms apply, since our our data model allows each object instance to be stored only in a single container at a time:

`union(A1,A1)` $\equiv$ `A1`
`intersect(A1,A2)` $\equiv \emptyset$
`minus(A1,A2)` $\equiv$ `A1`

Furthermore, we have:

- `union` is *associative* and *commutative*:
  `union(A1, union(A2,A3))`
  $\equiv$ `union(union(A2,A3),A1)`
  $\equiv$ `union(A1,A2,A3)`

- *Distributivity*:
  `intersect(union(A1,A3), union(A2,A3))`
  $\equiv$ `union(intersect(A1,A2),A3)`

$\equiv^2$ `A3`

With the above, we can eliminate all `intersect` and `minus` operators when flattening derived formulas into base formulas, thus reducing all base formulas to `union` expressions over cell references with their respective filter predicates.

5.2 Flattening Formulas

As we have defined earlier, the derivation of a formulas always forms an acyclic DAG structure. Flattening formulas into their base formulas is always possible when there are no cycles in the derivation. Thus, each newly created formula is directly checked for cycles when the cell entry is committed.

5.2.1 Union

The flattened formula expression `F(Z)` for cell `Z` with formula `union(S,T)` is:

```
F(Z) = union(F(S),F(T))
```

5.2.2 Intersection

We now construct the flattened formula expression `F(Z)` for cell `Z` with formula `intersect(S,T)`. Intersections are more complicated than unions.

The operation `concat(s,t)` concatenates the conditions of `t` to the conditions of `s`, thus returning a new container expression. For example:

`union(A1[species=fox],A1[day=2])` $\equiv$ `A1[species=fox & day=2]`

Also, in the following `cell(c)` denotes the container referred to by the formula `c`. For example if `c` is `A1[species=fox]`, then `cell(c)` is `A1`.

```
isect = {}
For each container expression s in F(S)
  For each container expression t in F(T)
    if(cell(s) == cell(t)) {
      f := concat(s,t)
      isect := union(isect,f)
    }
F(Z) := isect
```

[2]For base formulas only.

5.2.3 Minus

We now construct the flattened formula expression `F(Z)` for cell `Z` with formula `minus(S,T)`. We use a similar construction as in the case of intersections.

The operation `without(s,t)` returns a flattened expression `f`. For each condition `c` in `t`, we create a container expression `concat(s,!c)`, were `!c` is the logical negation of condition `c`. Then `f` is then the union over these container expressions. For example:

```
minus(A1[species=fox],A1[day=2 & species=deer])
≡ union(A1[species=fox & day!=2], A1[species=fox & species!=deer])
≡ union(A1[species=fox & day!=2 & species!=deer])
≡ A1[species=fox & day!=2 & species!=deer]
```

Then we have:

```
minus = {}
For each container expression, s, in F(S)
  For each container expression, t, in F(T)
    if(cell(s) == cell(t)) {
      f := without(s,t)
      minus := union(minus,f)
    }
  if(cell(s) does not equal cell(t) for any t in F(T)) {
    minus := union(minus,s)
  }
F(Z) := minus
```

5.2.4 Proofs of Constructions

Union: Suppose there is an object O_k in `union(S,T)`. Then $O_k \in S$ or $O_k \in T$. If $O_k \in S$, then O_k is in `F(S)`; if $O_k \in T$, then O_k is in `F(T)`. So O_k is in `union(F(S),F(T))`. Now suppose O_k is in `F(Z)`. Then O_k is in `F(S)` or `F(T)`. Then $O_k \in S$ or $O_k \in T$ and O_k is in `union(S,T)`.

Intersect: Suppose there is an object O_k in `intersect(S,T)`. Then $O_k \in S$ and $O_k \in T$. Then O_k is in `F(S)`, and O_k is in `F(T)`. If O_k is in `F(S)`, then it satisfies some formula expression s in `F(S)`. Similarly O_k is in some formula expression t in `F(T)`. As O_k can only be stored in a single container, we have `cell(s) = cell(t)`. Thus O_k satisfies `concat(s,t)`, which is in `F(Z)` as defined above.

Now suppose O_k is in `F(Z)`. Then O_k is in `concat(s, t)` for some formula expression s in `F(S)` and t in `F(T)`, where `cell(s) = cell(t)`. If O_k is in `concat(s,t)`, then $O_k \in s$ and $O_k \in t$, and therefore O_k is in `intersect(s,t)`.

Minus: Suppose there is an object O_k in `minus(S,T)`. Then $O_k \in S$ and $O_k \notin T$. Then O_k is in `F(S)`, and O_k is not in `F(T)`. If O_k is in `F(S)`, then it satisfies some formula expression s in `F(S)`. Similarly O_k does not satisfy any formula expression t in `F(T)`. If O_k does not satisfy any formula expression in `F(T)`, then O_k must not satisfy some condition in F', where F' is the list of all conditions for formula expressions f, such that `cell(s)` = `cell(f)`. Since O_k can only be stored in a single container, we have `cell(s)` = `cell(t)`. Then O_k must be in `minus(s,t)`, and in `minus(T)`.

Now suppose O_k is in `F(Z)`. Then O_k is in `minus(s,t)` for some formula expression $s \in S$ and $t \in T$, where `cell(s)` = `cell(t)`. If O_k is in `minus(s,t)`, then $O_k \in s$ and $O_k \notin t$, and therefore O_k is in `minus(s,t)`.

5.3 Intersect/Minus Operator Elimination

As we have seen before, each object is stored in only a single container at a time, or it is copied to another container and then forms a new object instance. That is, an intersection over different containers referenced by a base formula *must* yield an empty set. The following axiomatic transformation rules for base formulas make use of this assumption.

For example, minus over different containers in base formulas yields the left-hand container. Minus for the same container can be rewritten in the filter expression (ditto for union):

```
minus(A1[species=fox],A1[location=ridge])
 ≡ A1[species=fox & location!=ridge]

union(A1[species=fox],A1[location=ridge])
 ≡ A1[species=fox | location=ridge]

union(A1[species=fox & day!=2], A1[species=fox & species!=deer])
 ≡ A1[species=fox & (day!=2 | species!=deer)]
```

IMPORTANT: With these simple optimizations, we can effectively eliminate all `intersect` and `minus` operators in base formulas. Furthermore, each cell reference occurs at most once per `union` expression. This will enables us to rewrite the formula to a very efficient SQL statement, see Section 7.

6 Database Schema

$$images(ImageID, URI, Image)$$

(Image is the cached version of the photo, e.g., a BLOB in the database, or null for objects wrapping simple data types.)

$$tags(ColID, RowID, ImageID, Attr, Val)$$

(For containers with strings or simple data types, *ImageID* and *Attr* may be null.)

$$formulas(ColID, RowID, Formula)$$

(Formula entries would currently be stored as plain strings which get parsed against the above grammar)

7 Rewriting Formula Expressions to SQL Queries

Each filter predicate in a base formula can be directly translated to the WHERE clause of a SQL query over the above schema. For example, the formula

```
C1[species=fox & (location=ridge | temperature=50.4)]
```

can be rewritten to the following SQL query over the above schema:

```
select distinct t1.ImageID
from tags t1, tags t2, tags t3
where t1.ColID='C' and t1.RowID=1
and (
  (t1.Attr='species' and t1.Val='fox')
   and (
    (t2.Attr='location' and t2.Val='ridge')
     or
    (t3.Attr='temperature' and t3.Val='50.4')
   )
)
and t1.ColID=t2.ColID and t2.ColID=t3.ColID
and t1.RowID=t2.RowID and t2.RowID=t3.RowID
and t1.ImageID=t2.ImageID and t2.ImageID=t3.ImageID
```

`Union` operators then directly translate to union statements in SQL with multiple nested select statements, each corresponding to a different cell selection with a filter predicate. The default union behavior in SQL is duplicate-eliminating, too.