HyperFlow: An Integrated Visual Query and Dataflow Language for End-User Information Analysis

Dolev Dotan and Ron Y. Pinter
Department of Computer Science
Technion – Israel Institute of Technology
{dolevd, pinter}@cs.technion.ac.il

Abstract

We present HyperFlow, a novel visual language for information analysis that combines features from visual dataflow and visual query languages into a unified framework. HyperFlow is designed to make it easier for users to retrieve, filter, and manipulate information, using databases alongside e.g. web services, in a transparent, intuitive, reproducible and traceable manner. It allows users to visually design and execute information analysis processes in a single diagram. We present HyperFlow's constructs and describe the characteristics of a prototype interface we have implemented.

1. Introduction

Recent years have brought on an explosion of information in many domains. Considerable progress has been made in generic development of database and information retrieval technologies to handle new scales of data diversity and extent. Moreover, new fields – such as bioinformatics and business intelligence – emerged in an effort to create specialized tools that help retrieve this information, treat it in various ways, and mine it for interesting conclusions.

Still, many of the databases and analysis services provide only basic, under-expressive, and rigid user interfaces. For example, many information warehouses in the domain of bioinformatics provide forms-based interfaces which allow users to specify parameters and constraints over a single type of entity which they are interested in retrieving. They do not, however, allow users to use e.g. arbitrary joins in the conditions, not to mention the slightly more complex operations of retrieving a projection over a join, grouping, applying aggregate functions or using wrapped external functions.

Another major issue is that of data integration. The databases and analysis services are usually highly distributed. In addition, in active and open fields such as bioinformatics, such services are being constantly added and changed. Moreover, different databases and services use radically different formats for the same kind of data. Currently, the users themselves often have to bear the burden of finding the available databases and services, learning how to use each of them, and integrating their functions and formats into a unified information handling process.

In order to ease the orchestration of complex database queries and service executions into integrated information handling processes, we suggest leveraging the power and agility of visual languages. To this end, we present HyperFlow, a new visual language that represents the “marriage” of visual query languages (VQLs) and visual dataflow languages. The whole – resulting of this merger – is greater than the sum of its parts, as HyperFlow not only allows the simple combination of queries into workflows, but also provides increased expressive power, intuition, and simplicity, both in posing queries and in specifying workflows. HyperFlow is designed to support almost all constructs from SQL, OQL and RDF query languages, thus giving the user a unified language to most common databases available today.

The HyperFlow language lends itself to two modes of operation. In the first mode the user designs the workflow first and then executes it; in the second, the user performs interactive exploration of information (also called ad-hoc analysis). In the latter mode, a user can pose a query, review its results, and then decide on the next step, e.g. to pose a follow-up query or to execute a service or a pre-defined workflow. As an
added benefit, the workflow that is generated by a
session can be stored for traceability purposes and can
be re-used at a later point, even on different inputs.

In this paper we use examples from the domain of
bioinformatics to demonstrate the features and
advantages of the HyperFlow language. This domain is
uniquely complex in the amount and diversity of
available databases and services, and in the intricate,
semi-structured and highly-linked nature of its
schemas. However, there is nothing in the language that
is domain-specific; it can be used with any set of
ontologies, databases and services, in any domain.

The rest of the paper is organized as follows. In the
next section we walk through an example that
demonstrates some of the key features of HyperFlow.
In Section 3 we give a comprehensive, albeit informal
presentation of its various query and workflow
constructs. We then review the characteristics of an
interface that we have implemented. In Section 5 we
review related work. Finally, we conclude in Section 6
by exploring some future research directions.

2. HyperFlow by Example

A sample workflow, depicting a simple analysis
process, is shown in Figure 1. A researcher who has
isolated a DNA molecule in the lab wishes to study it
by comparing – in various ways – its sequence to
sequences in public databanks.

The user starts her analysis session with an execution
of the BLAST [24] service (in Figure 1a), which finds
alignments to similar sequences in the databank. Using
these alignments, she poses a follow-up query (b) that
requests the actual sequences that these alignments
represent. She further constraints the query to return
only those mammalian sequences that correspond to
alignments with more than 98% similarity. This is
depicted by a visual construct that – on one hand –
serves as a clear encapsulation of the query itself, and –
on the other hand – fits well within the overall dataflow
diagram.

With the relevant sequences from the databank at
hand, the scientist now proceeds to analyze them in
three different ways; this is immediately apparent from
the three arrows emanating from the results set box.
First, she decides to compare the original sequence she
isolated in the lab against each of the sequences she
retrieved. She does so by running (c) a global
alignment algorithm, once for each pair of the original
and retrieved sequences. Next she launches (d) a
service that gives a multiple alignment of the retrieved
sequences. Finally, to understand how the genes
encoded by the database sequences are controlled, the
scientist re-uses (e) a workflow she once prepared that
finds which transcription factors are common to all the
sequences in the group.

Figure 1. A simple bioinformatics analysis

Expressing this kind of analysis would be rather
difficult by using queries alone or dataflow alone. It is
the combination of the two paradigms, as well as the
introduction of constructs such as implicit iteration
(this and other novel language features will be
described in the next section), that make it possible to
construct this type of a session – let alone more
complex ones – in a simple and intuitive way.
Moreover, the pictorial rendering of the whole process
makes it both clear as well as well-defined at the same
time.

3. The HyperFlow Language

3.1. The Data Model and its Visual
Representation

At the base of HyperFlow lies an expressive object-
oriented data model, based on W3C's Semantic Web
Ontology Language OWL [18]. OWL provides
increased expressive power over the data models of
object-oriented and relational databases, adding
features such as individuals, enumerations (of literals
and individuals), local class restrictions, Boolean class
expressions, a property hierarchy and transitive
properties. We further extend OWL by introducing user-defined datatypes and collection classes.

Table 1 shows the visual representation of some of the basic ontology entities: classes, instances, and datatypes. Note that we differentiate between ontology classes and view classes, and between by-reference and by-value semantics, where applicable. This is useful in queries, where selecting an object by reference (see e.g. Figure 6) will yield – rather than the whole object – only its ID (URI, OID or primary key), accompanied by some textual description field when appropriate. This allows us to provide two-step searches, first retrieving IDs and later fetching the complete objects if needed, as is common in information retrieval over the Web.

<table>
<thead>
<tr>
<th>Entity Type</th>
<th>By Reference</th>
<th>By Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enumeration of instances</td>
<td></td>
<td></td>
</tr>
<tr>
<td>View class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Numeric datatype</td>
<td></td>
<td></td>
</tr>
<tr>
<td>String datatype</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Boolean</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Date/Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enumeration of numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enumeration of strings</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Properties of an entity are depicted using property edges, which are labeled (and blue – see the electronic version of this paper). Constraint nodes, such as in Figure 1, are connected to entities by (black) constraint edges. Finally, dataflow edges are dashed (and red).

Collections are represented, as is common in many VQLs, as rounded rectangles ("blobs") surrounding an entity icon which represents the collection's type. In case a property of the entity is shown, that property edge will leave the collection (see Figure 17). While this design decision admittedly prevents the user from seeing explicitly the nature of the collection's nesting, it was taken in order to prevent the unmanageable and unintelligible nesting of collection blobs.

![Diagram](image.png)

**Figure 2. Collections in HyperFlow**

The different collection types are depicted using two mechanisms (see Figure 2): In the first, an icon is used to differentiate an ordered list from an unordered collection. The second shows the collection's semantics with respect to duplications: By default, a collection uses the bag semantics (i.e. allows duplications). A collection with a set semantics (without duplications) has a thick outline.

### 3.2. Dataflow Constructs

As in all dataflow languages, HyperFlow analysis workflows are comprised of a network of modules, connected by dataflow edges (depicted as dashed red lines). HyperFlow further adds the concept of result-set boxes, which depict the type of information that results from each module and allows language features such as implicit iterations. In this section, we review each of those in turn.

#### 3.2.1. Modules

Modules are depicted as boxes that have input ports and output ports. These ports may be named and may have a type, which can be an ontology class, a primitive class, or a (possibly typed) collection class.

In HyperFlow, there are several types of modules: services, nested workflows, and queries. As we discuss in Section 3.3, some query constructs, such as group-by, order by, and aggregate functions, are also depicted as modules inside the query module. We treat them here as services since they can also be used as stand-alone services outside the query.

A service is depicted in the black-box fashion familiar from other visual dataflow languages: a box with the service name and possibly an icon. Workflows and queries can also be shown as black-boxes, which is useful for their reuse. In addition, HyperFlow provides a glass-box depiction of workflows and queries (such as the query in Figure 1), which allows them to be shown and edited in the same diagram as the outer process.

The input ports of services are predefined and dependent upon a given service profile (such as a WSDL or OWL-S document). In contrast, workflows and queries are user-defined; users add ports to workflows and queries while editing them.

Some services have many parameters, and usually only a few of those are relevant to the dataflow. For example, the BLASTP service takes in addition to the input sequence (which would usually come from the dataflow) several arguments such as: e-value, word-size, matrix name, and gap cost. Normally users would not want to specify these arguments in the workflow. Rather, they would prefer a properties window, in
which they can also accept some suggested defaults. To support this, HyperFlow enables **hiding** input ports. Each input port can be hidden by default and then **exposed** by the user, and vice versa.

### 3.2.2. Result Sets

In HyperFlow, each module is immediately followed by a result set box, depicted as a (yellow) box with a dashed outline. This box represents the database view that is created to hold the results returned from the module. A result set can store a single element or a collection of elements which can be ontology objects, datatype objects, tuple (i.e. query-defined) objects, and other collections. A notable exception is the result set of a construct-type query (see Section 3.3.9) which contains a graph of objects.

The type of objects in the result set is depicted inside the result-set box. In the case of tuple objects, the fields of the tuple are also shown. In the implementation, we show the result-set type while a query is being built. This lets the user have immediate visual feedback that can help him construct the query. Another form of feedback is the **collection size display**, which is added to collections in interactive exploration mode, after the results arrive (see Figure 3). Finally, it should be noted that like views in a database, result sets can be named, and this name is shown as a title on top of the result-set box.

Results sets serve as more than visual feedback. HyperFlow allows the user to **expand** (i.e. show) properties of the objects in the box. This provides the user fine control over what is passed to the next module: a user can connect the node representing an object’s field, instead of connecting the whole object. Another feature is the ability to conveniently send to the next module either the whole collection (see Figure 1d and e) or the collection’s members one by one in an implicit iteration.

### 3.2.3. Implicit Iteration

When the node that describes the type of members in a collection is connected to the input port of a module, this module is executed once for each collection member. This is called implicit iteration, and is demonstrated in Figure 1c. The result of the execution will be the collection of results from each execution, maintaining the order of the input collection (if applicable). In cases where each execution yields a collection, there is no automatic merging – the result will be a collection of collections.

Implicit iterations are more than just a depiction of the simple *for each* loop. They can also be used to go over multiple collections, if the receiving module has more than one input port. In these cases, the result will be an execution of the module for each combination that comprises one item from each collection (see Figure 3a). In addition, HyperFlow provides several **module execution modifiers** for controlling the iteration: For instance, when the input collections are ordered lists of the same length, the “i=j” modifier makes it possible to iterate over ordered pairs of list elements (see Figure 3b). Other modifiers (see Figure 4) allow iterating over various combinations of pairs of elements from a single collection.

![Figure 3. Iteration over two input collections](image)

![Figure 4. Iteration over one input collection](image)

Note that any other correlation between members of collections can be achieved by using a query which takes as input the group(s), and returns a collection of tuple objects that represent the needed correlation. The different fields of the tuple can then be connected to the input ports – a module which takes both inputs from the same result set is executed in a correlated mode.

HyperFlow also provides **input passing modifiers**. A module that is marked by such modifier (depicted as an unusually-colored output port) will return a tuple object consisting of the input and the output. If there is more than one input, there are two options: In the first (yellow output port – see Figure 5a) the inputs are shown as different fields of the tuple object. In the second (purple output port– see Figure 5b) the inputs are combined to a tuple of their own, and this tuple, with the output, forms the result object.

![Figure 5. Passing input through modules](image)
3.3. HyperFlow Queries

HyperFlow introduces a combination of declarative graph-based constructs with functional dataflow constructs. The declarative part allows users to specify the subgraph of the ontology which they are interested in, and add some constraints over the various object and datatype nodes. This is equivalent to the FROM and WHERE parts of an SQL/OQL query, as well as the use of path expressions in any construct of OQL. The other parts of a query – SELECT, GROUP-BY, ORDER-BY, functions, etc – are depicted using data-flow constructs, i.e. by using data-flow edges to send the selected entity nodes to the output port, while passing through transformation modules on the way (see e.g. Figure 13). Notice how this distinction nicely mirrors the linguistic distinction between the names of various SQL/OQL constructs; only the constructs which are written in imperative form (i.e. as commands) are depicted as processing modules.

3.3.1. Query Constraints. Constraints are shown in HyperFlow as orange hexagons containing the constraint symbol or word. They are connected to the schema node by means of unlabeled black edges. Examples of constraint usage are shown in Figure 6 and Figure 7. HyperFlow supports a natural usage of a wide variety of constraints, including constraints such as OQL’s collection inclusion constraint (Figure 7g) and SQL99’s type checking constraint (Figure 7i). The latter, and its sibling, are very useful when hierarchical controlled vocabularies are used in the ontology – such as the organism taxonomy hierarchy in Figure 1 and the Gene Ontology’s (GO) [1] Molecular Function hierarchy, in Figure 7.

Every constraint can be seen as a function whose return type is Boolean. If a user-defined function with such a signature exists in the databases, HyperFlow supports using it as a user-defined constraint in queries. One example of this is (j) in Figure 7. Other notable examples are some of the methods common in GIS databases, such as Contains() and Intersects().

3.3.2. Boolean Operators. HyperFlow allows combining constraints into Boolean expressions. This is done by using the AND, OR, and NOT operators, depicted by (blue) modifier nodes, to draw arbitrarily complex expression graphs. The various ways to use the operators are shown in Figure 8. Note that the AND modifier is usually superfluous, as AND is the default operator for combining constraints in a query. Also note the two different usage patterns of OR.

3.3.3. Joins. A join is created in HyperFlow by connecting two entity nodes (that do not represent fields of the same object) using a constraint (Figure 9a). Equi-joins can be created without using a constraint, as is demonstrated in Figure 9b. In addition, a Cartesian product is created whenever the nodes are not connected, as in (c). In order to allow naming specific fields of the resulting output class, HyperFlow...
provides a specialized module (the AS module that corresponds to the SQL aliasing construct AS), which the user places on the outgoing data flow edge (see Figure 9a).

3.3.4. Modifiers for All, Some, Transitivity, Optionally. (See Figure 10 and Figure 11):
- **all** and **some** – are like SQL’s ALL and EXISTS, respectively.
- **transitively** – is used to transitively follow properties which are defined as transitive.
- **optionally** – is used to define outer joins.

3.3.5. Selecting groups. HyperFlow has some built-in support for some of the queries which are natural to ask but usually difficult to phrase in an OQL query. For example, a user may ask to get all the genes that fit some criteria, fetching along with each gene ID also its collection of transcript RNA sequences. The way to do so in HyperFlow is to connect the required collection directly to the output port (see Figure 12). This saves the need to write a nested query, as in OQL.

3.3.6. Grouping. The HyperFlow analog of OQL’s GROUP BY statement is the Group A by B module. As its name suggests, it aggregates the results from the previous steps (that enter through port A) to groups, based on the value of the object or field that is connected to port B. The results are pairs, each consisting of the value of the aggregation criteria and the collection of objects that fit the criteria (partition, in OQL terms). The aggregated objects themselves can be of any type, including arbitrary projections, thus possibly creating a tree of objects.

HyperFlow makes it easy to filter out parts of the aggregated groups based on some aggregate function (i.e., to use the HAVING construct). In order to do this, the user instructs the system to show a result set inside the query, after the group-by module. The results displayed can then be connected via dataflow to any aggregate function module, whose result can again be selected, constrained, or made to flow to other modules.
3.3.7. Ordering. ORDER BY is depicted in HyperFlow by an "Order By" module. This module has two input ports – one for the input objects and one for the sort criterion. In order to sort by more than one criterion, extra criteria input ports can be added (see Figure 15).

3.3.8. Removing duplications. Another SQL construct which is depicted as a module is DISTINCT. We call the module "remove duplicates" to better describe it as a service (see Figure 16). This module, like group-by and order-by, can be used stand-alone in workflows.

3.3.9. Construct and construct graph. HyperFlow provides two modules that help users arrange their output: The construct module (see Figure 17) is an implementation of the "struct" constructor in OQL. It combines all its inputs as fields in a single output class. The resultant object can be named, just like in the AS module.

3.3.10. Functions and methods. Using the dataflow metaphor gives HyperFlow a natural method of depicting functions. This applies to built-in arithmetic and string functions as well as to aggregate functions and conversions (listset, flatten, element), set operators and user-defined functions. In order to clarify set operator semantics, the modules that depict them show icons with the respective Venn diagrams. Finally, we allow all query constraints, as well as Boolean operators, to be used as functions with a Boolean return value.
When coming to support object-oriented methods, we make a distinction between two types of methods, which we support in two different techniques. The first type contains methods with no argument. Since those methods depend solely on the object, we treat them as if they were ordinary properties, as is common in some object-oriented languages (e.g., properties in C#). We treat all other methods as what methods really are—functions that get as input, in addition to their stated arguments, also the object on which they operate.

3.3.11. Subqueries. A nested subquery is depicted in HyperFlow as a query module nested inside the main query module (see Figure 19). The results from the inner query are shown in the outer query, and can be related to specific nodes using constraints and modifiers. To create a correlated subquery, an input port is added to the subquery (see Figure 20). It accepts input from a node in the outer query, thus making explicit the fact that the inner query is run once for each outside node instance.

3.3.12. Query Naming and Recursion. HyperFlow allows queries and subqueries to be named. Named queries can be stored and reused at a later occasion. Moreover, a named query can be used as a subquery nested in any query—including itself—in a recursive fashion. An example of a recursive subquery is given in Figure 21. This type of query can be translated to SQL's WITH RECURSIVE statement.

4. The User Interface

In order to make HyperFlow a valuable tool in the hands of end-users, an easy-to-use interface must be implemented. Interaction with the HyperFlow environment is based on several familiar GUI features, such as drag and drop—for instance, queries often start by dragging a class icon from the Ontology Browser to a query box.

Another predominant feature is the use of context-sensitive popup-menus. Most of the interaction in HyperFlow actually starts with clicking on an item or the background of a box and choosing from the appropriate list of actions available. Examples for such actions are creating a new query and adding an input port to a query or a workflow.

Many of the actions on the context-sensitive menu are of a new type we call "connection expansion". For example, the context menu of a class node has a "Follow Property" submenu, which lists all the properties of the class. Each of these, in turn, opens another submenu with the possible ranges of the property (in OWL, a property can have more than one
range). The user chooses the range that he wants to include in the query and then indicates where to position the appropriate class or datatype node. The node is then inserted to the query, with the selected property edge leading to it from the original class node.

The same interaction type is used for adding constraints: the "Add Constraint" item in the context menu of a class or datatype icon will show a submenu containing constraints appropriate to it, which the user chooses and places. A click on the added constraint node will then allow the user to "Add a Right-Hand Value" in a similar way. Constraint nodes can also be easily connected to existing nodes (to create queries like Figure 6). The system prevents the user from creating illegal connections, leading to e.g. type errors.

The HyperFlow Editor is designed to make it easy to find and use services: \texttt{in} in the popup-menu of a result-set the system suggests the services applicable to it. It does so using semantic matchmaking between the result-set type to service profiles, written in the semantic web services description language OWL-S [19]. Note that because some services operate on an item and some on a collection, a class node and a collection blob will typically get very different service suggestions.

Another important feature that is implemented in HyperFlow is the ability to reuse a section of a workflow. This feature is called refactoring. The user selects the part of the workflow to be reused, and selects the "Refactor into New Workflow" action from the context menu. A new diagram is then opened, containing the selected sub-workflow, with the appropriate input and output ports. This diagram can then be saved and reused in subsequent sessions.

Our interface is implemented in Java as plug-ins for the open-source Eclipse Platform [21], using its Graphical Editing Framework (GEF) [22] as the foundation for the graphical editor. Ontology parsing and manipulation is done through the open-source library Jena [23].

5. Related Work and Comparison

Existing visual query languages (VQLs) use a variety of paradigms (for a review see [2]), perhaps the most prominent being that of graph-based queries, as in e.g. Gq[l3], QGraph[4], MMDDLQ[5], P/FDM[6], and HVQS[7]. Two other VQL paradigms are highly influenced by dataflow diagrams. One is filter-flow, which is implemented in the Kaleidoquery language [8], in which objects are filtered as they go down the flow; this allows for an effective and natural construction of Boolean expressions, but other constructs, such as result structuring, joins, and grouping are less intuitive. The other paradigm is to construct a dataflow of database operators, such as filter, project, join, order, and so on; this is done in DFQL[9], among others. This approach has the drawback that it makes no visual use of the database entity, and as such does not actually provide a way to see the query that is being built. VQLs also differ in the way and the extent to which they support features such as creating Boolean predicate expressions, ordering, grouping, and applying aggregate functions. A feature comparison table between HyperFlow and other languages is given in Table 2.

Several visual environments have been created recently to help scientists compose complex workflows. Notable research projects are Kepler[12], Taverna[13], and Triana[14], and pertinent commercial products are Pipeline-Pilot[15] and übertool[16]. As noted in e.g. [12], the characteristics of scientific workflows are different from those of general purpose workflows that can be described with emerging standards such as BPEL4WS. Primarily, scientific workflows are dataflow oriented, whereas business workflows (which drive the standards) are mostly control-flow oriented.

HyperFlow caters for scientific workflows, as is demonstrated in this paper for bioinformatics, as well as for any similar information handling process. As such, it is naturally a dataflow system. Its queries are depicted using a mixed approach that combines the graph-based and the dataflow paradigms. A somewhat similar approach was taken in the QUIVER VQL ([10], [11]); our proposal – we believe – is more expressive and more intuitive. Moreover, HyperFlow is unique in that it is not just a VQL but also a language for describing dataflow processes. It shares many of the basic constructs with other dataflow languages, but some of its important features (such as implicit loops) are supported by only a few languages, and others – such as native queries – are completely novel.

6. Conclusion and Further Work

We believe that HyperFlow's approach gives a way to specify both simple and complex queries in a natural way. In addition, being based on OWL, HyperFlow's ontology model provides expressive domain modeling. As far as we know, it is the first VQL to support such expressive ontologies.

Preliminary discussions with prospective users have assured us of the language's intuition and effectiveness in helping to construct information handling processes. We intend to perform a more comprehensive usability study of the language and of the interface in the future.
The HyperFlow language, as described here, contains only dataflow constructs. However, we are currently adding several control-flow constructs which will allow conditional execution, support for SQL's CASE statement, service orchestration, error-handling, and explicit loops. Another planned future activity is to provide a formal definition of the language.

Future work on the interface will include the implementation of new class-specific viewers, e.g. a viewer for protein 3D-structures. Each viewer will be associated with a class, just like the services, and the same matchmaking mechanism will be used to enable users to select from just the appropriate viewers. This will be a significant step in materializing our plan to provide scientists with a comprehensive environment to handle their information, in much the same way that programmers handle their work in an interactive development environment.

7. References
<table>
<thead>
<tr>
<th>Feature</th>
<th>HyperFlow</th>
<th>QUIVER</th>
<th>Kaleidoquery</th>
<th>Gql</th>
<th>QGraph</th>
<th>MDDQL</th>
<th>HVQS</th>
<th>VOQL</th>
<th>VOODOO</th>
<th>VQE</th>
<th>QBE</th>
<th>DFLQ</th>
<th>Iconic SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>B. model</td>
<td>OO (OWL-based)</td>
<td>OO (ODMG)</td>
<td>OO (ODMG)</td>
<td>OO (ODMG)</td>
<td>OO (ODMG)</td>
<td>OO</td>
<td>OO</td>
<td>OO</td>
<td>OO</td>
<td>OO</td>
<td>OO</td>
<td>OO</td>
<td>OO</td>
</tr>
<tr>
<td>V. paradigm</td>
<td>Graph Flow</td>
<td>Graph Flow</td>
<td>Filter Flow</td>
<td>Graph Flow</td>
<td>Graph Flow</td>
<td>Graph Flow</td>
<td>Graph Flow</td>
<td>Graph Flow</td>
<td>Frames</td>
<td>Frames</td>
<td>Frames</td>
<td>Data Flow</td>
<td>Iconic</td>
</tr>
<tr>
<td>Completely Visual</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Projection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓/NV</td>
<td>✓/NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
</tr>
<tr>
<td>Binary Constraints</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓/NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
</tr>
<tr>
<td>IN (constants)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓/NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
</tr>
<tr>
<td>Disjunction</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>P</td>
<td>✓/NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
</tr>
<tr>
<td>Negation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>P</td>
<td>✓/NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
<td>NV</td>
</tr>
<tr>
<td>Relationships</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Arbitrary joins</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓/NV</td>
<td>✓</td>
<td>✓</td>
<td>NV</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Outer Joins</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓/NV</td>
<td>✓</td>
<td>✓</td>
<td>NV</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Existential Quantifications</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Universal Quantification</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Transitive properties</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Group By</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Having</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>P</td>
<td>✓/NV</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Aggregate Functions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Binary Set Operators</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>P</td>
<td>✓/NV</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Collection Operators*</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>P</td>
<td>✓/NV</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Arithmetic Operators*</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓/NV</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>User-Defined Functions / Methods</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓/NV</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Order By</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓/NV</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Subqueries</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>P</td>
<td>✓/NV</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Distinct</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓/NV</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OF TYPE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓/NV</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Construct</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓/NV</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Construct Graph</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓/NV</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Output Field Aliases</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓/NV</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Recursion</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>P</td>
<td>✓/NV</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Closure</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>P</td>
<td>✓/NV</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Notes:** NV – supported in a non visual manner. P – Partially supported.
* – listtoset, flatten, element, etc.