DatumTron In-Memory Graph Database API

By
Ashraf Azmi
9/15/2015

Table of contents

- Introduction
- Using the code
- Background -- The Datum Universe
- The DatumTron API
  1. Initialization, saving and loading
  2. Adding Data
  3. Simple Data Queries
  4. Inheritance
  5. Representing Time
  6. Code as data
  7. Comparison with relational database
  8. Beyond SQL -- Data Mining
- Conclusions

Introduction

In this article, we introduce the DatumTron In-Memory Graph Database API. Data is represented as a directed acyclic graph of “datums” connected by “is” links. Because DatumTron API represents data at such a fundamental level, we are able to manipulate data generically. We discuss how inheritance, time, and code are represented in the graph.

We explain how to import an SQL database (Northwind) into memory as a DatumTron graph database. Then we show how to query, and mine the database in a fast in-memory graph using a simple set of operators. In DatumTron, finding all rows that have the same column value is achieved in constant time. For example, a query to get the customers who live in Paris is written in the following C# statement. Where CustomerCity is the datum representing the customer city column in the graph. This statement executes in an $O(1)$ time-complexity thanks to the referential nature of the graph and the use of hashing.

```csharp
var ParisCustomers = CustomerCity & "Paris" // O(1)
```

Writing multi-table queries is usually a short and clear statement. For example, the categories of products
ordered by customers who live in Tokyo, which in SQL, involves joining 5 tables, becomes a one line C# statement like:

```csharp
var TokyoCat = Category[Product[OrderDetail & (Order & (Customer & (CustomerCity & "Tokyo")))]]
```

This query also takes advantage of the graph and hashing to maximize execution speed.

We discuss how to mine the database for patterns, for example, is there a relation between where the customers live and the category of products they order? The following screenshot is from the datamining example at the end of the article.

--- Patterns: Any pattern in customer City and ordered product Category?
(City| México D.F.) -> 23% (CategoryName| Dairy Products) -- support = 17
(City| México D.F.) -> 25% (CategoryName| Beverages) -- support = 18
(City| Luleå) -> 25% (CategoryName| Beverages) -- support = 13
(City| Marseille) -> 22% (CategoryName| Seafood) -- support = 10
(City| Bracke) -> 26% (CategoryName| Beverages) -- support = 12
(City| San Cristóbal) -> 24% (CategoryName| Beverages) -- support = 11
(City| Cork) -> 21% (CategoryName| Seafood) -- support = 12
(City| Cork) -> 23% (CategoryName| Dairy Products) -- support = 13
(City| Toulouse) -> 32% (CategoryName| Beverages) -- support = 10
(City| Frankfurt a.M.) -> 25% (CategoryName| Dairy Products) -- support = 10
(City| Barquisimeto) -> 29% (CategoryName| Dairy Products) -- support = 10
(City| Cunewalde) -> 22% (CategoryName| Beverages) -- support = 19

--- Patterns: Any patterns in Employee selling product Category?
(Employee| 5) -> 26% (CategoryName| Dairy Products) -- support = 31
(Employee| 6) -> 23% (CategoryName| Dairy Products) -- support = 39
(Employee| 9) -> 22% (CategoryName| Beverages) -- support = 24
(Employee| 9) -> 23% (CategoryName| Dairy Products) -- support = 25
(Employee| 5) -> 26% (CategoryName| Dairy Products) -- support = 31
(Employee| 6) -> 23% (CategoryName| Dairy Products) -- support = 39
(Employee| 9) -> 22% (CategoryName| Beverages) -- support = 24

Using the code

The download package includes 3 folders; DatumImporter, Tutorial, and Datumtron. The code in this article is part of the Tutorial C# project. It will help, if you load the project in Visual Studio, run and debug it. Also included is the DatumImporter project, which you can easily modify to import your database. The DatumTron dlls are included in the DatumTron folder.

Background – The Datum Universe

The Datum Universe is a graph that represents data. Each node in the graph is a *datum* – which is the most abstract element of data. A datum has no content of its own, and is totally defined by links to
other datums. There is a single type of abstract links in the graph, called “is”. The Datum Universe is a Directed Acyclic Graph (DAG). The is links are directed up and there can be no circular links.

Primitive data types, for example, integers, strings, DateTimes, etc., are represented in their native representation as “objects”. A datum may have one (or zero) objects linked to it. In the DatumTron API, a datum that has an attached object is called katum. The “k” is to symbolize that it is a knowledge element.

As a graph, each node has multiple parents and multiple children. The set of katum parents are called Attributes and abbreviated by $A$. The set of katum children are called Instances and abbreviated by $I$.

In Figure 1, all nodes are katums since they have strings attached to them (“Male”, “Programmer”, “USA”, “Alex”, “Jim”, “Wei”, “Raj”). Of course, Male, Programmer, and USA are also instances of other katums; say Gender, Occupation and Country of residence. The individual programmers may have other attributes as well like instances of Salary. Since in this example, all individuals share the same three properties, we can classify them into one class as shown in Figure 2. The new element in the middle is a datum; it has no object, and is shown as a hollow circle. As a datum, it is totally defined by its links to other elements. By introducing this datum, we can reduce the number of links from $4 * 3$ to $4 + 3$. This reduces the size of the graph (Graph size is the number of edges). We can think of this datum as the US-Male-Programmers category. Datums represent pure classes and therefore, help in classification, data mining and machine learning.

In this document we focus mostly on katums. A katum must have at least one link to another element (datum or katum) and exactly one link to an object. The following code snippet is a rough sketch of the katum C# class.
For example; to represent the fact “apple is a fruit”, we create an apple katum that links to the fruit katum and attach the string “apple” to it. The strings “apple” and ‘fruit’ are the objects attached to the respective katums.

The following illustration shows a relational table that represents fruits and the corresponding Datum universe datum-pair representation.

<table>
<thead>
<tr>
<th>Relational Table</th>
<th>Datum Universe</th>
</tr>
</thead>
<tbody>
<tr>
<td>fruit</td>
<td>• fruit is thing</td>
</tr>
<tr>
<td>key</td>
<td>• color is thing</td>
</tr>
<tr>
<td>color</td>
<td>• apple is fruit</td>
</tr>
<tr>
<td>apple</td>
<td>• red is color</td>
</tr>
<tr>
<td>…</td>
<td>• apple is red</td>
</tr>
</tbody>
</table>

Notice that the table, the column, the row primary key, and field values, are all represented as katums, the distinctions between them come from how they are linked to each other in the graph. Figure 3 shows the corresponding graphical representation.

In the above example, fruit and red are attributes of apple, while apple is an instance of fruit, and red.

The first attribute of a katum is the default inheritance provider in DatumTron and is referred to as a0. In this case, fruit is the a0 attribute for apple, which means that apple inherits all the attributes of fruit by default.

**The Datum Universe vs. Entity-attribute-value and RDF models.**

The fundamental unit in EAV and RDF is the triple. For example, “The color of apple is red.” is represented as the triple: \((Apple, Color, Red)\)
In this triple, Apple is the entity, Color is the attribute of the entity and Red is the value of the attribute. On the other hand, in the Datum Universe, the fundamental unit is a pair. So the above triple is deconstructed into two pairs: (Apple, Red) and (Red, Color).

The advantage of a more granular model is that you have only two meta concepts to work with; the Datum and the "is" link. This allows us to manipulate data with a simpler set of operators and write more generic algorithms as will be shown in the rest of the article.

The DatumTron API

The DatumTron API is a .NET implementation of the Datum Universe model. DatumTron provides functions and operators to add and remove data to the graph, maintaining the graph’s integrity. Also provides the functions and operators necessary to query the data, and deduce new data not explicitly given. In the following sections we introduce some of the important features of DatumTron and show how to use and build on them. We also use Microsoft’s Northwind database to show how to convert relational databases to the Datum Universe and how to query multiple related tables using DatumTron ‘&’ operator.

Note on naming conventions: I opted for a minimalistic and language agnostic names, to provide a simple and consistent interface. In naming the classes 'datum' and 'katum', I chose to follow basic types like 'int' and 'string' using all lower case names. I may provide a C# oriented version of method and property names in the near future.

You can download the .NET dlls and the example solutions from www.DatumTron.com.
1. Initialization, saving and loading

There is a one-time initialization, by calling datum.setup. This call returns the root datum. The function takes an IAtum object that provides the concrete implementation of datums and katums. Interestingly, Atum is the name of the mythical god of creation, which fits nicely this creation context. IAtum keeps the door open for providing multiple implementations of datum and katum objects with different performance / memory profiles.

```csharp
katum thing = datum.setup(new atum());
```

You can save the database to datum file and load from that file. The following code extract shows loading the included northwind.datum file.

```csharp
katum northwindDB = katum.load(new atum(), @"..\..\northwind.datum");
```

2. Adding Data

Adding data involves creating a new katum and assigning attributes to it. A new katum must have a parent which is the base attribute `a0`. Also, a katum must have an attached object. In the following
examples, the attached objects are the string literals “color”, “taste”, etc. The base attribute a0 is the default provider of inheritance.

Creating a new katum is achieved by the “get” function which is also represented by the “|” operator. The “get” function means: find an existing katum, or create a new katum with this parent and this object. This is the only way to create new katums and add them to the Datum Universe graph. The get function insures that no two instances of a katum have the same object.

Notice that “thing” is the root defined in the previous code extract. Also, notice that the variable color holds a reference to a new katum which has the attribute katum thing, and the string object “color”.

```csharp
katum color = thing.get("color"); // New katum is created

katum c1 = color;                   // c1 holds a reference to the katum color
katum c2 = thing.get("color");
katum c3 = thing | "color";
// c3 is equal to c2, is equal to c1, and is equal to color
```

Now let’s define more katums.

```csharp
// define more katums using the | operator
katum taste = thing | "taste";
katum food = thing | "food";
katum fruit = food | "fruit";
katum apple = fruit | "apple";
katum mango = fruit | "mango";
katum red = color | "red";
katum sweet = taste | "sweet";
```

Assigning attributes to katums is achieved by using the “is” function (written as @is since “is” is a reserved C# keyword). The @is function adds attributes to the katum. The “set” function also adds an attribute to the katum, but overrides any similar attribute with the given value. For example, if we have “apple is red”, and we add “apple is green”, apple will have two colors red and green. On the other hand, if we have “apple is red”, and we do apple.set(color, green), apple will have one color; green. This is because “red” and “green” are both instances of “color”, so “red” will be overridden with “green”. This can also be achieved by the [] operator as shown below.

```csharp
// apple is red
apple.@is(red); // OR

// apple can have one color which is red
apple.set(color, red); // OR
color[apple] = red; // OR
color[apple] = color | "red"; // get color red if not created or create a new
```

Let’s define OK to do Debug.Assert() to keep the code snippets short and clear.
In order to remove data, we simply use the function “isnot”. The function isnot undoes the function is. Note that isnot can’t remove a0. Since a0 was added when creating the katum, it can only be undone by removing the whole katum.

apple.isnot(red); // apple is not red anymore
OK(!apple.Is(red));
apple.@is(red); // back to apple is red
OK(apple.Is red)

3. Simple Data Queries

Using the “find” function or the “&” operator, we can find katums by their attached object in an O(1) time complexity. If “find” fails, it returns null. Contrast this with, the “get” and “/” which will create a new katum if one is not found.

<table>
<thead>
<tr>
<th>method</th>
<th>operator</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>get</td>
<td></td>
<td>Create a new katum if one doesn’t exist already</td>
</tr>
<tr>
<td>find</td>
<td>&amp;</td>
<td>Find a katum, and if not found, return null</td>
</tr>
</tbody>
</table>

The question “Is red a color?” can be represented using the “Is” (Upper case I) function as “red.Is(color)?” Another variant on the “Is” function is the “Isa” function which only returns true if the attribute is the a0 attribute of this. So apple.Is(red) is true while appl.Isa(red) is not true and apple.Isa(fruit) is true.

katum red1 = color.find("red");  OK(red1 == red);
katum red2 = color & "red";    OK(red2 == red);
katum blue = color & "blue";   OK(blue == null);

OK(red.Is(color)); // Is red a color ?
OK(red.Isa(color)); // Isa -> color must be the first attribute of red

OK(color.of(apple) == red);
OK(color[apple] == red);

In the example of color[apple] = red, color is a “Property” of apple, red is an “Attribute” of apple, and red is an “Instance” of color. In a.of(b), the of function finds an attribute of b that is an instance of a. The attribute found is not necessarily a direct instance of a. The of function starts the search from the direct attributes of b, and keeps moving up towards a. In the above example, red just happens to be a direct attribute of apple and a direct instance of color.
At this point if you want to focus on database query, you can skip to “Comparison with relational database”

4. Inheritance

Inheritance is based on the transitive relationship; \textit{if a is b and if b is c then a is c}. In the Datum Universe, for a katum \( k \) and attribute set \( \{a_0, a_1, \ldots, a_n\} \), \( k \) is \( a_0 \) and \( k \) is \( a_1 \), \ldots, and \( k \) is \( a_n \). Now, if \( a_0 \) is \( a_{00} \) and \( a_1 \) is \( a_{11} \), \ldots, \( a_n \) is \( a_{nn} \), then \( k \) is \( a_{00} \), \( k \) is \( a_{11} \), and \( k \) is \( a_{nn} \) by transition. For performance reasons, in DatumTron’s \textit{of} function, the built in inheritance uses only \( a_0 \) by default for inheritance. In the following code extract, \textit{since fujiApple is an apple and since an apple is red, then fujiApple is red}.

```plaintext
katum fujiApple = apple | "fujiApple";
OK(color[fujiApple] == red); // inherited from apple
```

To explore multiple inheritance, DatumTron has the enumerator \textit{Ax}, which iterates over all attributes and their attributes.

Another form of inheritance is shown in the following code extract. In this case, we know that \textit{fujiApple’s color is darkRed}. If we ask “Is the color of \textit{fujiApple} red?” we get true, since \textit{darkRed is red}. Note that \textit{color[fujiApple]} is not equal to \textit{red} but only “is red”.

```plaintext
katum darkRed = red | "darkRed";
apple.@is(darkRed);
OK(color[fujiApple] == darkRed);
OK(color[fujiApple] != red);
OK(color[fujiApple].is(red));
OK(red[fujiApple] == darkRed);
```

5. Representing Time

As we have seen so far, any katum can have a set of instances represented by the \textit{I} set. Each instance katum has its own object, in our examples so far it has been mostly strings. So apple is an instance of fruit and \textit{fujiApple} is an instance of apple. If we like to change an attribute, we can use the \textit{set} function to modify the value. However, if some attributes do naturally change over time, we use the “\textit{now}” function. In the following code extract, \textit{fujiApple.now( pricePerPound | 1.3 )} creates a new instance of \textit{fujiApple}, attach an object to it of type “\textit{time}” containing the current DateTime value. Copies all the attributes from \textit{fujiApple}, then sets \textit{fujiApple} pricePerPound to 1.3. The copied instance is known to
DatumTron as a Time Instance and contains the previous state of its a0 katum (in this case fujiApple) up until the time stamp attached to it.

If we ask what is the pricePerPound of fujiApple, we get the latest value which is 1.3. If we need to access previous time instances of fujiApple, we use the “at” function or “[ ]” operator as shown in the following code extract. Time instances are treated just like any instance, with the exception of inheritance. Inheritance of attributes skips a0, since a0 contains the “now” attribute values.

```plaintext
katum pricePerPound = thing | "pricePerPound";
  fujiApple.set(pricePerPound | 1.5);
  fujiApple.now(pricePerPound | 1.3); // price changing over time
  fujiApple.now(pricePerPound | 1);

OK(pricePerPound[fujiApple] == 1); // current time instance of fujiApple
OK(pricePerPound[fujiApple.at(0)] == 1.5); //OR just use [0]
OK(pricePerPound[fujiApple[0]] == 1.5);
OK(pricePerPound[fujiApple[1]] == 1.3);
OK(fujiApple.countI == 2); // number of instances under fujiApple
```

DatumTron provides countI property which is equivalent to I.Count but relies on the concrete implementation for a fast O(1) time complexity. This gives the instance count, whether they are regular or time instances.

### 6. Code as data

The object attached to a katum can be of type “function”, such that when the value of the object is requested, the function is executed and its value is returned. For example, in the following code extract

```plaintext
katum banana = fruit | "banana";
katum weight = thing | "weight";

apple.set(weight | 1);
mango.set(color | "green", weight | 0.9);
banana.set(color | "yellow", weight | 1.1);

fruit.set(weight | AvgFunc); // <---- set the weight of fruit to a function object
OK(weight[fruit] == 1); // average of 1, .9, 1.1
```

Because the weight of fruit is set to a function object AvgFunc, when the value of weight of fruit is requested, the function AvgFunc is executed. The AvgFunc is a user provided function and has to have the function signature as follows.
The body of the AvgFunc is shown in the following code extract. It uses the I enumerator to iterate over the instances of k and calculate the value of the numberAttribute at each instance, take the object attached as a double value and calculate the average.

```csharp
public delegate object function(katum property, datum item);

static object AvgFunc(katum numberAttribute, datum k)
{
    if (k == null || numberAttribute == null) return null;
    return k.I.Average(i => numberAttribute[i].asDouble);
}
```

Notice that the AvgFunc is totally generic, it doesn’t know about fruits or weights. It just calculates the average of an attribute of all instances of a katum. However, other less generic functions may look for specific attributes, like quantity and price, to calculate total price.

Another way to embed code is to have an object that implements the IFunction interface. There is another example for this in the Tutorial code for calculating Total Price in an Order Detail table – look for TotalPrice class.

7. Comparison with Relational Database

In order to show real world queries, we use the Microsoft Northwind sample database. You can find the database diagram including tables and columns as well as download Northwind from https://northwinddatabase.codeplex.com/

The following diagram shows the relationships among the various tables in the database but more geared towards the Datum Universe.
Starting at the bottom of the diagram,

- The Order Detail has Order and Product among its attributes. This also means that each Order instance has Order Detail instances. Furthermore, each Product instance has Order Detail instances that represents all Order Detail items from this product.
- The Order has among its attributes, Shipper, Customer and Employee. Again, this means that an instance of Shipper, Customer, or Employee has instances that are orders.
- The Product has Category and Supplier among their attributes. As above, instances of Supplier and Category have instances that are products, representing products in a specific category and products supplied by a specific supplier.

Understanding this diagram is important in creating queries whether in SQL or DatumTron.

The following code snippet is part of the included DatumImporter project. It shows the `ImportNorthwind` function. You can replace `TableDescription` objects with your database table description.
The DatumImporter is a reference implementation of a relational database importer that creates a "Table" and "Column" katums under the root. Each table has an instance under the Table katum. Each table also has an instance under the Column katum which represents its set of columns. For example, the Employees table in the Northwind database is represented by "Employee" katum. Each Primary key in the Employees Table is represented by an instance under the Employee katum. In order to get the field value we simply use column[key], for example lastName[1] gives the last name of the employee with employeeID equals 1.

Figure 5 shows the two top katums; Table and Column. The Left tree shows the Table katum expanded which has 11 instances representing tables in Northwind. The Employee table is expanded showing an instance for each key. Employee 3 is expanded to show its instances. Since an Order has an Employee as one of its columns attributes, we should expect to see Orders as instances of an Employee. Order 10309 is expanded showing its related Order Detail katums. Again, since an Order Detail has an Order as one of its attributes, then we should find a set of Order Detail instances.

The right tree in Figure 5 shows the Column katum expanded showing a katum representing each table’s column set. Expanding the column set of Product, we see the various columns like Product key, ProductName, and Discontinued attribute. Expanding the Discontinued katum shows two instances False and True. Under “True” we find all product instances that are discontinued. If we look at the instances of a product, we find the set of Order Detail instances that had this product. This is because Product is an attribute of column in Order Details {Table}.

```csharp
static katum ImportNorthwind(IAtum atum, string rootName, string NorthwindConnectionString)
{
    var NorthwindScheme = new DatabaseScheme(new []{
        new TableDescription("Customers", "Customer", new [] {"CustomerID"}),
        new TableDescription("Orders", "Order", new [] {"OrderID" },
            new [] {
                new ForeignKey("CustomerID", "Customers"),
                new ForeignKey("EmployeeID", "Employees"),
                new ForeignKey("ShipVia", "Shippers" )},
        // Other tables...
    });
    return Importer.ImportDatabase(
        atum, rootName, NorthwindScheme, NorthwindConnectionString);
}
```
As shown in the code sample below, to get the Employee katum, we use *Table & “Employee”*.

The relational database importer uses Primary and Foreign keys to establish the relationships between tables. The database is saved as datum universe file. In the following code snippets, we show loading the file and storing table and column katums in variables.
To get a specific Employee column, we first get the katum representing the set of columns of the Employee table, and then we need to & with the column name. Either in one step or in two steps as follows:

EmployeeCols = Column & “Employee”;
EmployeeLastName = EmployeeCols & “LastName”;
//OR
EmployeeLastName = Column & “Employee” & “LastName”;

**Iterating over table rows using the “I” Instance Enumerator**

DatumTron API provides a number of Iterators over the datum universe graph. Iterators come in pairs; one iterates in the down direction and one in iterates in the up direction. For now, let’s consider the Instance iterator – abbreviated as I.

For example, Employee.I iterates over all employee instances (rows in the relational table). There are supporting properties and functions; countI which gives the count of instances, and the hasI which returns true if a specific instance exists in I. There is also the at(index) which returns the instance at the specific index. Notice that countI, hasI and at, relay on the specific implementation of katums and usually perform in O(1). An additional convenience function is doI(action, condition) which invokes an action on every instance that satisfies the given condition.
To query using a where clause, we can just use the `IEnumerator` `Where` function or we can use the `doI`, specifying a condition. Both are equivalent.

```csharp
// Iterate over all employees
OK(Employee.I != null);
Console.WriteLine("All Employees");

foreach (katum employee in Employee.I)
    Console.WriteLine(employee);

Console.WriteLine("All Employees again");
// Shorter alternative
Employee.doI(Console.WriteLine);
```

To query using a where clause, we can just use the `IEnumerator` `Where` function or we can use the `doI`, specifying a condition. Both are equivalent.

```csharp
// List employees older than 50 years old
var EmployeeBD = EmployeeCols & "BirthDate";

foreach (var employee in Employee.I.Where(
    e => EmployeeBD[e] < DateTime.Today.AddYears(-50)))
    Console.WriteLine(employee);

// Shorter alternative
Employee.doI(Console.WriteLine, e => EmployeeBD[e] < DateTime.Today.AddYears(-50));
```

The `&` operator has three overloads as shown in the table below.

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters, example, explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td><code>katum &amp; object © katum</code></td>
</tr>
<tr>
<td></td>
<td>color &amp; “red”</td>
</tr>
<tr>
<td></td>
<td>Finds a <code>katum</code> that is an instance of color and has the object “red”. If no <code>katum</code> with the string “red” is found, the expression evaluates to <code>null</code>.</td>
</tr>
<tr>
<td>andI</td>
<td><code>katum &amp; katum © IEnumerable&lt;katum&gt;</code></td>
</tr>
<tr>
<td></td>
<td>programmer &amp; male © {Alex, Jim, Raj, Wei}</td>
</tr>
<tr>
<td></td>
<td>Enumerates instances of both <code>programmer</code> and <code>male</code></td>
</tr>
<tr>
<td>andI</td>
<td><code>Katum &amp; IEnumerable&lt;katum&gt; © IEnumerable&lt;katum&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>Order &amp; LondonCustomers</code></td>
</tr>
<tr>
<td></td>
<td>Enumerates over orders of all customers in London. Enumerate instances of each London customer that are also instance of <code>Order</code>.</td>
</tr>
</tbody>
</table>
In the following example, we find the column that represents the Employee’s Country, and then we find the instance representing USA. The & operator internally calls find which locates an instance that has the given object. Mainly using hashing, this operation has time complexity of O(1). We then iterate over the instances of the “USA Employees” katum and print them.

```csharp
// Iterate over USA employees
katum USAEmployees = EmployeeCols & "Country" & "USA"; // O(1)
OK(USAEmployees!​=null);
Console.WriteLine("USA Employees");
foreach (katum employee in USAEmployees.I)
    Console.WriteLine(employee);
```

A one line version of above is shown in the following code extract.

```csharp
(EmployeeCols & "Country" & "USA").doI(Console.WriteLine);
```

**Multi-Table Queries – No Join is needed.**

Queries involving multiple tables are handled in SQL with Join statements. In the equivalent Datum Universe representation, the relationships are embedded in the graph. We can achieve the same using & operator, and the [] operator. The [] operator has three overloads shown in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters, example, explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>at</td>
<td>katum[int] → katum</td>
</tr>
<tr>
<td></td>
<td>color[0] → red</td>
</tr>
<tr>
<td></td>
<td>Gets the color instance at index 0. Used for regular as well as time instances.</td>
</tr>
<tr>
<td>of</td>
<td>katum[katum] → katum</td>
</tr>
<tr>
<td></td>
<td>color[apple] → red</td>
</tr>
<tr>
<td></td>
<td>Gets the attribute of apple that is an instance of color. Note that, if color has a function attached, it is executed instead.</td>
</tr>
<tr>
<td>of</td>
<td>katum(IEnumerable&lt;katum&gt;) → IEnumerable&lt;katum&gt;</td>
</tr>
<tr>
<td></td>
<td>color[fruit.I] → {red, green, yellow}</td>
</tr>
<tr>
<td></td>
<td>gets the color of all fruit instances; color[apple], color[mango], color[banana]. By default, returns distinct values.</td>
</tr>
</tbody>
</table>
In the following code snippet, since orders have an employee property, `Employee[order]` gives the employee who took the order. We then get that employee’s last name using `EmployeeLastNameCol[...].`

```csharp
// List the order number and employee’s last name
// that have placed orders to be delivered in Belgium.
var EmployeeLastNameCol = EmployeeCols & "LastName";
var BelgiumOrders = OrderCols & "ShipCountry" & "Belgium"; // O(1)

foreach (var order in BelgiumOrders.I)
    Console.WriteLine("{0} by {1}", order, EmployeeLastNameCol[Employee[order]]);
```

In the following query, we use the Shipper, Customer, and Order tables. Recall from the Northwind diagram in Figure 4, that an instance of Shipper has instances that are orders representing orders shipped by a shipper. This means that if we have a specific shipper like SpeedyExpressShipper and iterate over its instances, we will find the orders shipped by it. In the following code fragment, the expression `SpeedyExpressShipper.I` gives us the instances of SpeedyExpressShipper. The expression `Order & SpeedyExpressShipper.I` gives us the instances which are orders (in case there are other types). Similarly, we get orders bought by customers living in Buenos Aires. Finally, we intersect the two sets of orders to get orders that are sent by the company Speedy Express to customers in Buenos Aires.

```csharp
// Give the order id, employee id and the customer id for
// orders that are sent by the company ‘Speedy Express’ to customers
// who live in Buenos Aires.
var SpeedyExpressShipper = ShipperCols & "CompanyName" & "Speedy Express"; // O(1)
var BuenosAiresCustomers = CustomerCols & "City" & "Buenos Aires"; // O(1)

var O1 = Order & SpeedyExpressShipper.I; // orders sent by Speedy Express
var O2 = Order & BuenosAiresCustomers.I; // orders bought by Buenos Aires customers

foreach (var order in O1.Intersect(O2))
    Console.WriteLine("{0} {1} {2}", order, Employee[order], Customer[order]);
```

In the following query, we get the products bought or sold by Londoners. Keeping in mind the database diagram, let’s breakdown the expression `OD1 = OrderDetail & (Order & (Customer & customerCityLondon));` The most inner part is `(Customer & customerCityLondon)` this is equivalent to `customerCityLondon.I` but insures that we take instances that are customers only. The expression `Order & (Customer &
customerCityLondon)` gets the orders of those customers. The full expression gets all of the Order Detail instances in these orders. Similarly, we get Order Detail instances from orders made by Londoners employees. At the end, we use `Product[orderDetail]` to get the product from the orderDetail.
8. Beyond SQL – Data Mining

In the previous query, we listed orders shipped by a specific company to a specific city. What if we want to see if there are any patterns in the product ordered and the cities they were ordered from? For example, do orders from Berlin focus on some products while orders from Madrid focus on other products? So we are looking for patterns in this form “if customer city is city1 then product ordered is product1” This also needs to include how many instances that support this pattern and what is the probability associated.

In data mining, this is called Frequent Pattern Mining. Let’s define “ifSide” to be condition “customer city is city1” and “thenSide” to be the conclusion “product ordered is product1”. Let’s also define “Support” to be the number of instances of orders that satisfy both the condition and the conclusion. Lastly, let’s define “Confidence” to be the probability that the ordered product is product1 given that customer city is city1.

The following code extract shows how we can go about this in the Datum Universe.
Notice the parameter `distinct: false` overrides the default value of `true`. We use this because we will count the occurrences of each distinct value to see if there are prevalent values.

Here are the supporting functions `FreqDict` and `PrintPattern`.

```csharp
const int minSupport = 10;  // minimum 10 orders
const int minConfidence = 20;  // minimum 20% probability

// Customer City and Product Category
var categoryName = Column & "Category" & "CategoryName";

foreach (var city in (Column & "Customer" & "City").I)
{
    var categories = Category[Product[
            OrderDetail & (Order & (Customer & city)),
            distinct: false ],
        distinct: false];
    int total;
    var freqDict = FreqDict(categories, out total);
    PrintPattern(city, categoryName, freqDict, total, minSupport, minConfidence);
}

private static Dictionary<T, int> FreqDict<T>(IEnumerable<T> items, out int total)
{
    // How many times each item is found in items
    var dict = new Dictionary<T, int>();
    total = 0;
    
    foreach (var item in items)
    {
        if (!dict.ContainsKey(item)) dict.Add(item, 1);
        else dict[item]++;
        total++;
    }
    
    return dict;
}

private static void PrintPattern(katum condition, katum conclusion, Dictionary<katum, int> freqDict, int total, int minSupport, int minConfidence)
{
    foreach (var item in freqDict.Keys)
    {
        var support = freqDict[item];
        var confidence = (support*100)/total;

        if (support >= minSupport && confidence > minConfidence)
            Console.WriteLine("\{0\} \{1\}% \{2\} -- support = \{3\}".,
                condition, confidence, conclusion[item], support);
    }
}
```
The advantage of the Datum Universe representation is that it is fundamental enough to allow us to create generic reasoning algorithms. For example, let us generalize the previous example so that we can find similar patterns just by stating what we want.

First step, let’s generalize the

```csharp
var categories = Category[Product[
    OrderDetail & (Order & (Customer & city)), distinct: false ],
    distinct: false];
```

Let’s call this type of pattern the “V-Pattern” because we go on a V path on Figure 4. That is, we start at the top from the customer, we move down to the order, then to the Order Detail, and then we move up to Product, then to Categories.

```csharp
var categories = VPatterns(
    city, new[] { Customer, Order, OrderDetail }, new[] { Product, Category });
```

Where VPattern is as follows:

```csharp
private static IEnumerabelekatum> VPatterns(
    katum condition,  
    katum[] ifPath, 
    katum[] thenPath)
{
    var eset = ifPath[0] & condition;
    for (var i = 1; i < ifPath.Length; i++)
        eset = ifPath[i] & eset;

    varelist = thenPath[0][eset, distinct: false];
    for (var i = 1; i < thenPath.Length; i++)
        elist = thenPath[i][elist, distinct: false];

    return elist;
}
```

We can further generalize to have another VPatterns function that accepts an `IEnumerable<katum>`, minimum support, and minimum probability and returns the patterns. Each pattern will be in the form of a `Tuple<katum, katum, int, int>` for example, `<City|Toulouse, Category|1, 10, 32>` means that 32% of the city of Toulouse order items are from product category 1, and this pattern is supported by 10 instances.
We can then call this function as follows:

```csharp
var patterns1 = VPatterns(Column & "Customer" & "City",
    new[] {Customer, Order, OrderDetail},
    new[] {Product, Category},
    10,
    20);

foreach (var pattern in patterns1)
    Console.WriteLine("{0} --> {1} support = {2} probability = {3}%",
                      pattern.Item1, pattern.Item2, pattern.Item3, pattern.Item4);
```

Now to look for a relationship between employees and category of products they sell, we can use the new VPatterns function as follows:

```csharp
var patterns2 = VPatterns(Table & "Employee",
    new[] {Order, OrderDetail},
    new[] {Product, Category},
    10,
    20);

foreach (var pattern in patterns2)
    Console.WriteLine("{0} --> {1} support = {2} probability = {3}%",
                      pattern.Item1, pattern.Item2, pattern.Item3, pattern.Item4);
```

The `VPatterns` function is now totally generic. Given the conditions source (Customer city or Employee), a path down and a path up, and minimum support and probability, the function can detect patterns of
this “type” on any database. Moreover, one can write more code to discover the path down and path up given the conditions source and the conclusion domain (in these examples, Category). This code gives the console output in the introduction.

9. Conclusions

We summarize the advantages of using DatumTron graph representation into performance gains, ease of querying complex database, native understanding of time, representing code as data, and providing a platform for data mining and machine learning.

Performance:

As a graph of nodes, we can find a node, hold a reference to that node, and then use it to find other nodes. Finding an instance katum based on its object value, is achieved in approximately constant time regardless of the number of katums in the graph.

\[
\text{Parent.find(object)} \rightarrow \text{instance in } O(1)
\]

Parent & object

For queries where we use equality, for example; Get Employees where employee city = Tokyo; a typical relational database scans all employees and select the ones where the city column has the value ‘Tokyo’. This has time complexity of \( O(n) \) where \( n \) is the number of employees. In the Datum Universe, we find the ‘Tokyo’ katum from the City katum in \( O(1) \). The instance set of Tokyo has the Tokyo Employees.

For queries where we apply a condition on a single column, for example; Get Employees where salary > 100,000; a typical relational database scans all employees and apply the condition on each employee. Again, this has time complexity of \( O(n) \) where \( n \) is the number of employees. In the Datum Universe, we scan the salary instances for the ones > 100,000. This is achieved in \( O(m) \) where \( m \) is the number of unique salary instances -- which is less than the number of employees. Each salary instance has the set of employees who get that salary.

For queries that involve multiple tables, relational database joins these tables and conducts the query. A modern relational query server creates a temporary data structure to improve the performance of the Join query for example; using Hash join and Merge join algorithms. In the Datum Universe, there is no need for this whole process; the graph represents data in an already “hashed” form.

The performance gains are not only as a result of in-memory processing but also the result of the Datum Universe graph representation itself, as well as the DatumTron API implementation and use of Hashing.

Ease of use:
DatumTron API use of a small set of operators allows writing statements that are clear and concise. This is especially true when multiple “tables” are involved. For example, products ordered by customers who live in München are

```javascript
var MunchenProducts = Product [ OrderDetail & (Order & (Customer & customerCityMunchen))];
```

Contrast this with the equivalent SQL query statement which contains 4 joins.

**Temporal:**

DatumTron API has built-in understanding of the concept of time. The ‘now’ operator allows you to change data with the understanding that it is time changes not corrections or permanent modifications. The ‘now’ operator creates a new Time Instance, and the ‘at’ operator allows you to retrieve time instances by their time index. Time instances are regular katums with a ‘time’ object attached. As any other instance, you can find a time instance by its attached time object. Using these simple operators, we can build more complex temporal facilities to analyze change due to the passing of time in specific periods of time and to recognize trends.

**Code as Data:**

Storing and executing code in DatumTron is seamlessly integrated in the API. Code is represented by objects of type ‘function’ or any class implementing the ‘IFunction’ interface. These objects are attached to katums as any other object type. The ‘of’ operator recognize these two types and executes the attached function. These katums have attributes and instances like any other katums. On one hand, this gives executable code access to the data in the graph and on the other hand, allows for selecting code to execute based on its attributes.

**Intelligence:**

This is the distinctive advantage of the Datum Universe representation and the DatumTron API. Representing data as a structure of one fundamental element (datum) with one fundamental relationship (is) enables us to create more complex but generic inference logic and data structures.

The simplicity of the building blocks of the Datum Universe, lends itself to building more complex data structures. We have seen how relational database can be represented and how this simplified and improved the time complexity of querying the database. Tables provide a good visual or mental picture of large amounts of similar data. Other forms of visualizations for example, graphs and trees provide more mental pictures of network or hierarchical oriented data. The Datum Universe can be the underlying representation of data while tables, networks, and trees can be the human mental model or the higher level visualization of data.

Inheritance is inferencing new information about datums by looking at their parents attributes – that is by looking ‘up’ in the graph. By looking ‘down’ at instance attributes, we can generalize and theorize attributes of the parents. If all instances of apple in the Datum Universe are red, then we can theorize
that an apple is red. We have seen how these generalizations are qualified by ‘support’ and ‘probability’ factors. We also briefly mentioned at the beginning (Figure 2) the use of datums to group katums that share a set of common attributes. DatumTron API has additional operators to ‘induce’ and manipulate datums in order to classify the graph. These features combined, provide the basics for data mining, classifications and machine learning.