Database Technology Evolution II: Graph Database Language

Malcolm Crowe
Emeritus Professor, Computing Science
University of the West of Scotland
Paisley, United Kingdom
Email: Malcolm.Crowe@uws.ac.uk

Fritz Laux
Emeritus Professor, Business Computing
Reutlingen University
Reutlingen, Germany
Email: Fritz.Laux@reutlingen-university.de

***Abstract*— This paper reviews the changes in database technology represented by the incorporation of property graphs and associated language in the International Standards Organization (ISO) standard 9075 (Database Languages – Standard Query Language SQL) and the current development of the draft international standard ISO 39075 (Database Languages – Graph Query Language GQL), and presents an implementation of the resulting combined technology in a single relational database management system. These developments continue a trend towards integrating conceptual modeling design into the physical database.**

Keywords—typed graph model; graph schema; relational database; implementation; information integration.

I. Introduction

For many years, the process of database implementation has included a conceptual data modeling phase, and this has often been supported by declarative structures using annotations or attributes, as reported in our previous contribution [1]. Graph models have become popular for this purpose, originally in relation to social networks, and numerous graphical database products such as Neo4j have applied these in many domains.

The growth in the use of graph models has led to the development of standards including the publication of ISO 9075-16: Property Graph Queries [2], and the imminent emergence of a draft international standard for GQL [3], [4]. These developments draw on experience with commercial graph database products and envisage a clear convergence at the conceptual level between graph-based and relational database management, while GQL remains a separate standard.

Our previous work has recommended the use of a Typed Graph Model (TGM) for conceptual modeling [5], with the help of additional data types in the Relational Database Management System (RDBMS) specified using metadata. In this paper we present an open-source RDBMS implementation that is able to perform graph creation and pattern matching including repeating patterns and also aligns well with the draft international GQL standard.

The plan of this paper is to review the new implementation details in Section II. Section III presents an illustrative example, and Section IV provides some conclusions.

II. Implementation in the Relational Database Schema

The implementation of a typed graph modelling system can build on the user-defined type mechanism of an RDBMS. Node and edge types can have special columns for leaving and arriving properties, which should have additional automated support from the RDBMS. It should be possible to convert between standard types and node/edge types and rearrange subtype relationships. These tables can be equipped with indexes, constraints, and triggers in the normal ways.

Then, if every node type or edge type corresponds to a single base table containing the instances of that type, one way to build a graph is to insert rows in these tables. But a satisfactory implementation needs to simplify the tasks of graph definition and searching. Most implementations add CREATE and MATCH statements, which we describe next, and indicate how they can be implemented in the RDBMS.

A. *Graph-oriented syntax added to SQL*

The typical syntax for CREATE sketches nodes and edges using additional arrow-like tokens, for example:

```
[CREATE (:Person {name:'Fred Smith'})<-[:Child]-
(a:Person {name:'Peter Smith'}),
(a)-[:Child]->(b:Person {name:'Mary Smith'})
-[:Child]->(:Person {name:'Lee Smith'}),
(b)-[:Child]->(:Person {name:'Bill Smith'})]
```

Without any further declarations, this builds a graph with nodes for Person and edges for Child, as in Figure 2(b). There is already a standard syntax for this in [2]. But an RDBMS engine can and should without further declaration also build base tables for Person and Child with columns sufficient to represent the specified properties, and indexes to support the edge structure.

The MATCH query can contain unbound identifiers for nodes and edges, their labels and/or their properties, which are bound by searching the database. This also has a standard syntax in [2], but in this section we indicate how it can be integrated into the SQL data manipulation language DML:

MatchStatement = MATCH Match {',' Match} [WhereClause] [Statement] [THEN Statements END].
Match = (MatchMode [id '='] MatchNode) {'|' Match}.

In this syntax, Statement(s) and WhereClause are as in ordinary SQL. The first part of the MATCH clause has an optional MatchMode (see below) and one or more graph expressions, which in simple cases appear to have the same form as in the CREATE statement.

MatchNode = '(' MatchItem ')' {(MatchEdge|MatchPath) MatchNode}.
MatchEdge = '-[' MatchItem '->' | '<-' MatchItem ']-' .
MatchItem = [id | *Node*_Value] [GraphLabel] [Document | WhereClause] .

In all cases, the execution of the MATCH proceeds directly on the tables, without needing auxiliary SQL statements. The MATCH algorithm proceeds along the node expressions, matching more and more of its nodes and edges with those in the database by assigning values to the unbound identifiers. If we cannot progress to the next part of the MATCH clause, we backtrack by undoing the last binding and taking an alternative value. If the processing reaches the end of the MATCH statement, the set of bindings contributes a row in the default result, subject to the optional WHERE condition.

In this way, the MATCH statement can be used (a) as in Prolog, to verify that a particular graph fragment exists in the database, (b) to display the bindings resulting from the process of matching a set of fragments with the database, (c) to display a set of values computed from such a list of bindings, or (d) to perform a sequence of actions for each binding found. In case (d) no results are displayed, as the MATCH statement has been employed for its side effects. These could include further CREATE, MATCH or other SQL statements, or assignment statements updating fields referenced in the current bindings.

Following the forthcoming GQL standard, repeating patterns are supported by the MATCH statement (see [8]):

MatchPath = '[' Match ']' MatchQuantifier .
MatchQuantifier = '?' | '*' | '+' | '{' int , [int] '}' .
MatchMode = [TRAIL|ACYCLIC| SIMPLE] [SHORTEST |ALL|ANY] .

MatchMode controls how repetitions of path patterns are managed in the graph matching mechanism. A MatchPath creates lists of values of bound identifiers in its Match. By default, binding rows that have already occurred in the match are ignored, and paths that have already been listed in a quantified graph are not followed. The MatchMode modifies this default behaviour: TRAIL omits paths where an edge occurs more than once, ACYCLIC omits paths where a node occurs more than once, SIMPLE looks for a simple cycle. The last three options apply to MatchStatements that do not use the comma operator, and select the shortest match, all matches or an arbitrary match.

The implementation of the matching algorithm uses continuations to control the backtracking behavior. Continuations are constructed as the match proceeds and represent the rest of the matching expression.

The MATCH statement can be used in two ways. The first is make the dependent Statement a RETURN statement that contributes a row to a result set for each successful binding of the unbound identifiers in the MATCH, for example,

```
MATCH ({name:'Peter Smith'}) [()-[:Child]->()]+
(x) RETURN x.name
```

will yield a list of the descendants of Peter Smith. (See Figure 2(a).)

Without using RETURN or any dependent statements, the result of a MATCH statement is the list of bindings. The following example has two columns, one for each of the unbound identifiers p and x, but p will be an array with an element for each iteration of the pattern.

```
MATCH ({name:'Peter Smith'}) [(p)-[:Child]->()]+
({name:x})
```

The results are shown in Figure 2(a), which also shows all of the statements needed to build and display this small example, including two lines for authentication for browser access to the database, and two for replacing the default primary key ID. A feature of the implementation described in this paper is the lack of structural clutter.

B. *Graph versus Relation*

The nodes and edges contained in the database combine to form a set of disjoint graphs that is initially empty. Adding a node to the database adds a new entry to this set. When an edge is added, either the two endpoints are in the same graph, or else the edge will connect two previously disjoint graphs. If each graph in the set is identified by a representative node (such as the one with the lowest uid) and maintains a list of the nodes and edges it contains, it is easy to manage the set of graphs as data is added to the database.

If an edge is removed, the graph containing it might now be in at most two pieces: the simplest algorithm removes it from the set and adds its nodes and edges back in.

The database with its added graph information can be used directly in ordinary database application processing, with the advantage of being able to perform graph-oriented querying and graph-oriented stored procedures. The normal processing of the database engine naturally enforces the type requirements of the model, and also enforces any constraints specified in graph-oriented metadata. The nodes and edges are rows in ordinary tables that can be accessed and refined using normal SQL statements. In particular, using the usual dotted syntax, properties can be SET and updated, and can be removed by being set to NULL.

C. *Database Design by Example*

From the above description of the CREATE statement, we can see that this mechanism allows first versions of types and instances to be developed together, with minimal schema indications. The MATCH statement allows extension of the design by retrieving instances and creating related nodes and edges.

If example nodes and edges are created, the DBMS creates suitable node and edge types, modifying these if additional properties receive values in later examples. Since transactions are supported, tentative examples can be explored and rolled back or committed. Alter statements can change names, enhance property types and modify subtype relationships, and the SQL Cast function can be used to parse the string representation of a structure value. The usual restrict/cascade actions are available, and node and edge types can have additional constraints, triggers, and methods. As each node and edge type has an associated base table in the database, the result of this process is a relational database that is immediately usable.

As the TGM is developed and merged with other graphical data, conflicts will be detected and diagnostics will help to identify any obstacles to integrating a new part of the model, so that the model as developed to that point can be refined. The SQL ALTER TABLE and ALTER TYPE statements, together with a metadata syntax, allow changes to the model to be performed automatically, e.g., to enforce expectations on the data.

An extreme case of this occurs where a graph has been created using the server's autokey mechanism for primary keys, and the analyst has identified a more suitable numeric or string-valued key. A single ALTER TABLE statement can install this as the new primary key and the change automatically propagates to the edge types that attach to the node type in question. The previous primary key remains as a unique key but can later be dropped without losing any information. See Figure 2 and Figure 3 below.

Other restructuring of node types can be performed with the help of the CAST function, which can be used to parse complex types from strings, array and set constructors, and UNNEST. Node and edge manipulations can also be performed by triggers and stored procedures.

The points covered in the above section already go a long way towards an integrated DBMS product that supports the TGM. The resulting TGM implementation inherits aspects such as transacted behavior, constraints, triggers, and stored procedures from the relational mechanisms, since Match and Create statements are implemented as Procedure Statements. The security model in the underlying RDBMS, with its users, roles, and grants of privileges also applies to the base tables and hence to the graphs. Node and edge types emerge as a special kind of structured type. It is thus a relatively simple matter to support view-mediated remote access and object-oriented entity management. Nodes and edges are entities and the same access and Multiple Version Concurrency Control (MVCC) models in our previous work [11] transfer with little trouble into the new features.

Our database server implementation has for years generated classes for C#, Python or Java applications corresponding to versioned database objects. This leads to object-oriented application programming, where node and edge types correspond to classes whose instances are nodes and edges (see Figure 2(c) for a C# example). The Match and Create statements can be used (a) for SQL clients in commands and prepared statements, (b) in the generated C#, Java or Python and the widely used database connection methods ExecuteReader and ExecuteNonQuery, or (c) in JavaScript posted to the web service interface of the database server.

The normal processing of the database engine naturally enforces the type requirements of the model, and also enforces a range of constraints specified in graph-oriented metadata. In particular, using the usual dotted syntax, properties can be SET and updated, and can be removed by being set to NULL.

As the TGM is developed and merged with other graphical data, conflicts will be detected and diagnostics will help to identify any obstacles to integrating a new part of the model, so that the model as developed to that point can be refined.

It is natural to expect a user interface that displays a graphical version of the property graph. Figure 3 shows that Pyrrho's HTTP service can draw a picture of a portion of a graph starting at a given node.

III. AN EXAMPLE

Examples for a graph structure usually choose social networks. We want to show that the TGM is equally suitable for Enterprise Resource Planning (ERP) and other business systems. As a non-trivial example, we have chosen a commercial enterprise, which buys parts and products, resells the purchased products or assembles products from purchased parts and sells these value-added products. It does not develop and construct products from raw material but adds some value to parts or assembles some products to form systems.

The data model is suitable for a customer-supplier ordering system and comprises 3 company divisions or departments: sales (green), stock (blue), and procurement (red). These are framed in Figure 1(a) with a green dashed line for sales data, with blue for stock data, and red for procurement or purchase. The graph schema is visualized using UML notation, which allows specifying the cardinality of the edges. The correspondence between Typed Graph elements and the UML is shown in Table I.

The sales division consists of Customer nodes with properties CustNo, Name and Address. The Name and Address might as well be structured data types for first- and last name resp. street, ZIP code, and city. The CustOrder node mainly comprises OrdNo, the (redundant) CustNo, order date Date and the order total Sum in Euros. The CustOrder contains 1 to many order detail lines of OrderPos, which consist at least of the order quantity as property. According to the semantics of the TGM the edge arrows signify the reading direction of the edge type. In the case of "belongs_to" the reading direction is from OrderPos to CustOrder.

All other necessary properties for an order line (e.g., partNo, PartName, UnitPrice) could be determined by following the edges of the model to the Part, Stock, and CustOrder node. In Figure 1 (a), only the nodes Customer and CustOrder are showing exemplified properties. More properties are maintained in a real situation, e.g., planned delivery, shipping date, etc. for a customer order. The same applies to all other nodes, e.g., unit and quantity discount for parts.

The procurement division mirrors the sales model structurally and comprises supplier, the purchases (SupplOrd, PurchPos) and the supplier catalogue. Purchase- and Sales divisions have connections to the stock management.

The central node of the stock model is the Part node who distinguishes between purchased parts (PurchasedParts) and in-house products (InHouseProduct) modelled as subtypes of Part. We have a BOM structurally represented as a recursive edge "part_of" on the part nodes. The BOM forms a tree structure with the product at the top. The product is made up recursively of components (composed parts) and finally of single parts. The stock itself is represented as a node with properties like number of parts, reservations, and commissions. A stock node is linked to a part and a storage location. This allows knowing exactly which part is located at a certain location in the warehouse.

Figure 1 (b) gives a high-level view of the scenario. Such abstractions are important for complex graphs in order to keep the model manageable. CASE tools that support zoom-in and zoom-out functions would be beneficial to assist the graph modelling.

The syntax of the above presented example ERP model will be presented in the following subsection. Multiline statements are enclosed in square brackets.

A. *Syntax of the ERP example*

First, we start with the sales graph (green schema), followed by the supplier (red schema) and stock division (blue schema), and finally the three divisions are linked by the edge types “serves”, “supplies”, “canSupply”, “orders”, and “from”.

```
// sales division
[CREATE
(a:Customer {CustNo:1001, Name:'Adam', Address:'122, Nutley Terrace, London, ST 7UR, GB'} ), // Customer
// …
(f:Customer {CustNo:1006, Name:'Eddy', Address:'72, Ibrox Street, Glasgow, G51 1AA, UK'} ), // customer without order
 (o1:CustOrder {OrdNo:2001, CustNo:1001, Datum:DATE'2023-03-22', SummE:211.00} ), // CustOrder
// …
(o8:CustOrder {OrdNo:2008, CustNo:1002, Datum:DATE'2023-04-24', SummE:808.00} ),
(op1:OrderPos {Quantity:4, Unit:'piece'} ), // OrdPos
// …
(op18:OrderPos {Quantity:10, Unit:'piece'} ),
(a)<-[:ORDERED_BY]-(o1), // each order was ordered by exactly 1 customer
(a)<-[:ORDERED_BY]-(o6),(a)<-[:ORDERED_BY]-(o7),
(b)<-[:ORDERED_BY]-(o2),
//…
(o1)<-[:BELONGS_TO]-(op1), // each orderPos belongs to exactly 1 order
// …
(o8)<-[:BELONGS_TO]-(op9), // and an order has at least 1 orderPos
// …
(o8)<-[:BELONGS_TO]-(op18)]
```

```
// supplier division
[ CREATE
(a:Supplier {SupplNo:101, Name:'Rawside Furniture', Address:'58 City Rd, London , EC1Y 2AL, UK'} ),
// …
// SupplOrd
(o1:SupplOrd {OrdNo:2001, SupplNo:101, Datum:DATE'2023-01-11', "Sum€":260.00} ),
// …
// OrdPos purchase details
```

```
(op1:PurchOrd {PosNo:1, Quantity:4, Unit:'piece'} ),
// …
// (Supplier)<-[:SUPPLIED_BY]-(SupplOrd)
(a)<-[:SUPPLIED_BY]-(o1), // each order was ordered by exactly 1 Supplier
// …
// (SupplOrd)<-[:IS_POS_OF]-(OrdPos)
(o1)<-[:IS_POS_OF]-(op1), // each PurchPos belongs to exactly 1 order
// …
```

```
(o1)<-[:IS_POS_OF]-(op7), // and an order has at least 1 PurchPos
//…
// SupplCatalog
(sc11:SupplCatalog {SupplNo:101, SPartNo:'sp1', description:'Hammer handle, Wood (ash), Weight:100 g', unit:'piece', unitPrice:2.00}), //P15
//P16
// …
(sc46:SupplCatalog {SupplNo:104, SPartNo:'sp6', description:'Shelf spruce, Color: white, Weight:6 kg, Size:60w x180h cm', unit:'piece', unitPrice:20}),
// (Supplier)-[:HAS]->(SupplCatalog)
(a)-[:HAS]->(sc11), (a)-[:HAS]->(sc12), (a)-[:HAS]->(sc13), (a)-[:HAS]->(sc14), (a)-[:HAS]->(sc15), (a)-[:HAS]->(sc16),
(b)-[:HAS]->(sc21), (b)-[:HAS]->(sc22), (b)-[:HAS]->(sc23), (b)-[:HAS]->(sc24), (b)-[:HAS]->(sc25)]
```

```
// stock division
// create Part types
create type Part as (PartID char ,Designation char, Color char, Weight char, Size char) nodetype
// PurchasedPart
create type PurchasedPart under Part as
(PreferredSupplNo int, sumOrderedThisYear currency, discountPrice currency)
// InHouseProduct
create type InHouseProduct under Part as
(ProductionPlan char, producedThisYear int, manufacturingCosts currency)
[CREATE
(a1:Location {LocationNo:10011, Aisle:1, Shelf:'left A', Rack: 'A1'} ), // Location
// …
(l:Location {LocationNo:10111, Aisle:2, Shelf:'left A', Rack: 'A1'} ), // Location without parts
//Part will be filled implicitly
// PurchasedPart
(p1:PurchasedPart {PartID:'P01', Designation:'Wallplug',Material:'Fiber', Color:'grey', Weight:'6 g', Size:'12 cm', PreferredSupplNo:103, sumOrderedThisYear:2000, discountPrice:'0.04 €' }), //p1 Wallplug
(p5:PurchasedPart {PartID:'P05' ,Designation:'Metal nail', Material:'Metal', Color:'grey', Weight:'2 g', Size:'A 50 x2.2 mm', PreferredSupplNo:102, sumOrderedThisYear:10000, discountPrice:'0.005 €'}), //p5 Metal nail
// …
(p30:PurchasedPart {PartID:'P30' ,Designation:'Degreasing liquid', Material:'benzine', Color:'clear', Weight:'100 g', Size:'100 ml bottle' ,
    PreferredSupplNo:101, sumOrderedThisYear:150, discountPrice:'1.80 €'}), //p30 Degreasing liquid
// InHouseProduct
(p2:InHouseProduct {PartID:'P02' ,Designation:'Power plug', Color:'white', Weight:'30 g', Size:'dia 5 cm ',
ProductionPlan:'P02 Power plug', producedThisYear:1000, manufacturingCosts:'2.50 €'}),
// …
(p28:InHouseProduct {PartID:'P28' ,Designation:'Tableleg', Material:'Metal',Color:'Silver', Weight:'1 kg',Size:'80w x120l cm', ProductionPlan:'P28 Tableleg', producedThisYear:160, manufacturingCosts:'7.00 €'}),
// Stock
(s1:Stock {PartID:'P02', LocationNo:10011, available:55, commissioned:20, reserved_until:DATE'2023-09-22'} ),
// …
(s34:Stock {PartID:'P30', LocationNo:10101,
available:30, commissioned:5,
reserved_until:DATE'2024-09-21'} ),
 //BOM
(p2)<-[:IS_Part_OF {no_of_components:2}]-(p12)<-
[:IS_Part_OF {no_of_components:1}]-(p13),
(p3)<-[:IS_Part_OF {no_of_components:1}]-(p14),
// …
(p26)<-[:IS_Part_OF {no_of_components:1}]-(p23),
// Links: Parts<-Stock->Location
(p1)<-[:stocked]-(s33)-[:at]->(i3),
),
// …
(p30)<-[:stocked]-(s34)-[:at]->(k)]
```

```
// linking together the 3 divisions
// links between Customer and Stock
// (OrderPos)-[:ORDERS]->(Part) (4)
[  match  (o1:CustOrder  {OrdNo:2001})<-[:BELONGS_TO]-
(op1:OrderPos {Pos:1} ),  (p2:Part {PartID:'P02'})
 create (op1)-[:ORDERS]->(p2) ]  // P02 Power plug
[  match  (o2:CustOrder  {OrdNo:2002})<-[:BELONGS_TO]-
(op2:OrderPos {Pos:1} ), (p10:Part {PartID:'P10'})
create (op2)-[:ORDERS]->(p10) ] // Rubber glue
// …
// (OrderPos)-[:FROM_]->(Stock) (5)
[  match  (o1:CustOrder  {OrdNo:2001})<-[:BELONGS_TO]-
(op1:OrderPos  {Pos:1}  ),   (s1:Stock  {PartID:'P02',
LocationNo:10011})
 create (op1)-[:FROM_]->(s1) ] // Power plug
[  match  (o2:CustOrder  {OrdNo:2002})<-[:BELONGS_TO]-
(op2:OrderPos  {Pos:1}  ),  (s30:Stock  {PartID:'P10',
LocationNo:10083})
create (op2)-[:FROM_]->(s30) ] // Rubber glue
// …
// links between Supplier and Stock
// (PurchPos)-[:SUPPLIES]->(PurchasePart) (2)
[  match  (o1:SupplOrd  {OrdNo:2001}  )<-[:IS_POS_OF]-
(pp1:PurchPos    {PosNo:1}    ),    (p18:PurchasePart
{PartID:'P18'}),
 create (pp1)-[:SUPPLIES]->(p18) ] // P18 Splint pin
[  match  (o2:SupplOrd  {OrdNo:2002}  )<-[:IS_POS_OF]-
(pp2:PurchPos    {PosNo:1}    ),    (p10:PurchasePart
{PartID:'P10'}),
 create (pp2)-[:SUPPLIES]->(p10) ] // P10 Rubber Glue
// …
// (SupplCatalog)-[:CAN_SUPPLY]->(PurchasePart) (3)
[     match     (sc11:SupplCatalog     {SupplNo:101,
SPartNo:'sp1'}), (p15:PurchasePart {PartID:'P15'}),
create  (sc11)-[:CAN_SUPPLY]->(p15)  ]   //P15  Hammer
handle
[               match               (sc12:SupplCatalog
{SupplNo:101,SPartNo:'sp2'}),       (p16:PurchasePart
{PartID:'P16'}),
 create (sc12)-[:CAN_SUPPLY]->(p16) ] //P16 Table top
// …
// links between Supplier and Customer
// (PurchPos)-[:SERVES]->(OrderPos) (1)
```

```
[  match  (so1:SupplOrd  {OrdNo:2001,  SupplNo:101})  <-
[:IS_POS_OF]-(pp1:PurchPos  {PosNo:1}),  (o1:CustOrder
{OrdNo:2001})<-[:BELONGS_TO]-(op1:OrderPos {Pos:1})
create (pp1)-[:SERVES]->(op1) ] //P16 Table Top
[  match  (so1:SupplOrd  {OrdNo:2001,  SupplNo:101})  <-
[:IS_POS_OF]-(pp7:PurchPos  {PosNo:2}),  (o2:CustOrder
{OrdNo:2002})<-[:BELONGS_TO]-(op12:OrderPos {Pos:2})
29.     create (pp7)-[:SERVES]->(op12) ] // P17 Table
Frame
// ..
```

There are opportunities here to alter some of these types to implement some of the comments in the model. Table I summarizes the schema objects (node and edge types) of the ERP graph schema and Figure 1 presents the TGS in graphical form using UML notation.

IV. Conclusions and Future Work

The purpose of this paper was to report on a successful mechanism for graph modeling, creation, and pattern-matching in an RDMS. The software is available on Github [8] for free download and use and is not covered by any patent or other restrictions.

The current "alpha" state of the software implements all of the above ideas. The test suite includes simple cases that demonstrate the integration of the relational and typed graph model concepts in Pyrrho DBMS.

Future work will include meeting the requirements of successive drafts of the GQL standard and enhancing the typed modeling features.

References

[1] M. Crowe and F. Laux, "Database Technology Evolution", IARIA International Journal on Advances in Software, vol. 15 numbers 3 and 4, 2022, pp. 224-234, ISSN: 1942-2628

[2] ISO 9075-16 Property Graph Queries (SQL/PGQ), International Standards Organisation, 2023.

[3] N. Francis et al., A Researcher's Digest of GQL. 26th International Conference on Database Theory (ICDT 2023), Mar 2023, Ioannina, Greece, doi:10.4230/LIPIcs.ICDT.2023.1, pp. 1-22. https://hal.science/hal-04094449 [retrieved: 18 October 2023]

[4] https://www.GQLStandards.org, October 4, 2023 – GQL status update [retrieved 18 October 2023].

[5] F. Laux and M. Crowe, "Information Integration using the Typed Graph Model", DBKDA 2021: The Thirteenth International Conference on Advances in Databases, Knowledge, and Data Applications, IARIA, May 2021, pp. 7-14, ISSN: 2308-4332, ISBN: 978-1-61208-857-0

[6] M. Crowe and F. Laux, "Graph Data Models and Relational Databe Technology", DBKDA 2023: The Fifteenth International Conference on Advances in Databases, Knowledge, and Data Applications, IARIA, March 2023, pp. 33-37, ISSN: 2308-4332, ISBN: 978-1-68558-056-8

[7] M. Crowe, PyrrhoV7alpha, https://github.com/MalcolmCrowe/ShareableDataStructures [retrieved: Sept 2023]

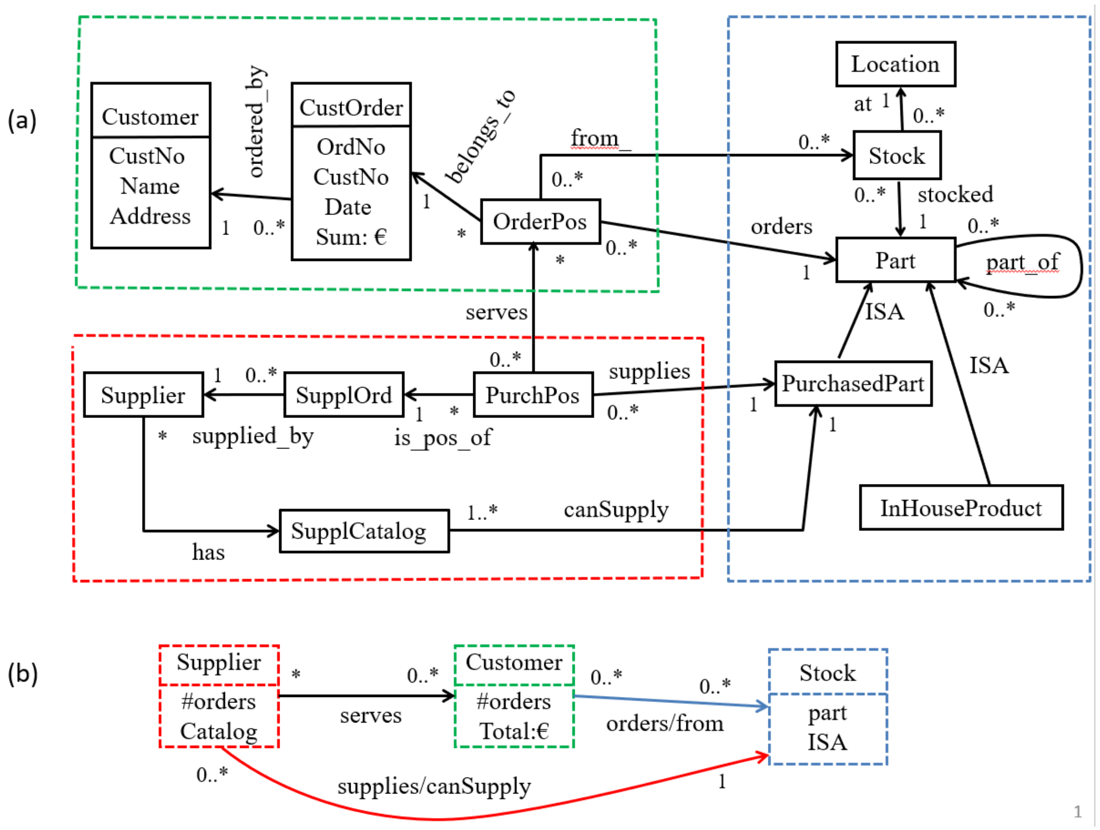
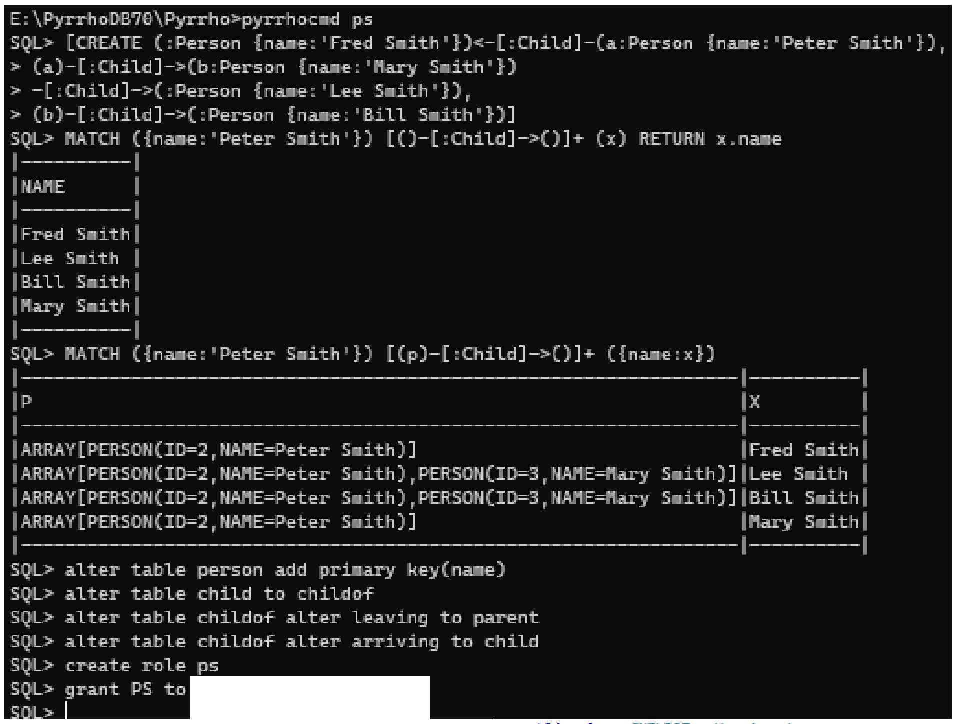


Figure 1. Example TGM of a commercial enterprise showing two levels of detail

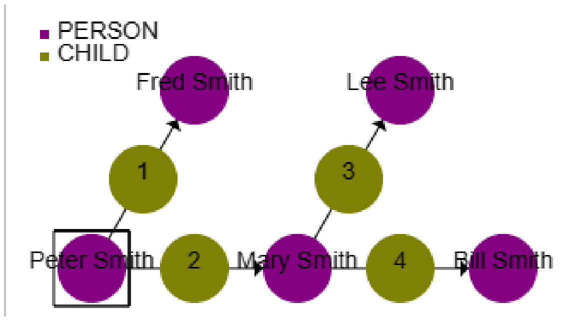
TABLE I. NODE AND EDGE TYPES IN AN EXAMPLE DATABASE (RELATIONAL DESCRIPTION)

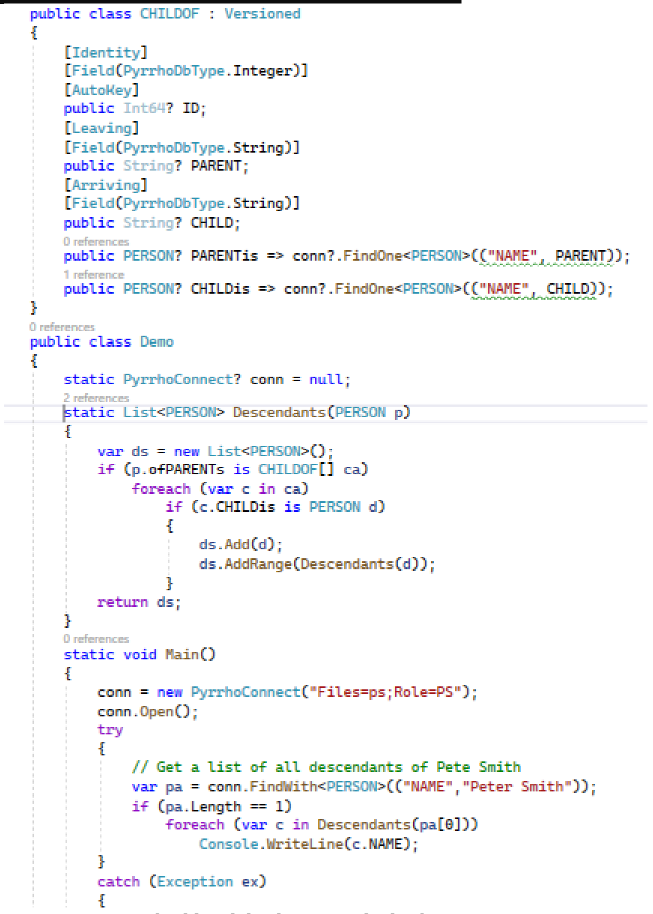
Type name	Informal Description	SuperType
Customer	(CustNo, Name, Address)	
CustOrder	(CustNo, Datum, OrdNo, Summ€)	
OrderPos	(Id, Quantity, Unit)	
Location	(LocationNo, Reihe, Shelf, Rack)	
PurchasePart	(PartID, Designation, Material, Color, Weight, Size)	Part
InHouseProduct	(PartID, Designation, Material, Color, Weight, Size)	Part
Stock	(PartID, LocationNo, Available, Commissioned, Reserved_Until)	
Supplier	(SupplNo, Name, Address)	
SupplOrd	(OrdNo, SupplNo, Datum, Sum€)	
PurchPos	(PosNo, Quantity, Unit)	
SupplCatalog	(SupplNo, SPartNo, Desription, Weight, Unit, unitPrice)	

Type name	Leaving	Arriving	Other properties
Ordered_by	CustOrder	Customer	
Belongs_to	OrderPos	CustOrder	
Is_Part_Of	Part	Part	No_of_components
Stocked	Stocked	Part	
At	Part	Location	
Supplied_by	SupplOrd	Supplier	
Is_Pos_of	PurchPos	SupplOrd	
Has	Sypplier	SupplCatalog	
Orders	OrderPos	Part	
From_	OrderPos	Stock	
Supplied	PurchPos	ParchasePart	
Can_Spply	SupplCatalog	PurchasePart	
Serves	PurchPos	OrderPos	



```
E:\PyrrhoDB70\Pyrrho>pyrrhocmd ps
SQL> [CREATE (:Person {name:'Fred Smith'})<-[:Child]-(a:Person {name:'Peter Smith'}),
> (a)-[:Child]->(b:Person {name:'Mary Smith'})
> -[:Child]->(:Person {name:'Lee Smith'}),
> (b)-[:Child]->(:Person {name:'Bill Smith'})]
SQL> MATCH ({name:'Peter Smith'}) [()-[:Child]->()]+ (x) RETURN x.name
|----------|
|NAME      |
|----------|
|Fred Smith|
|Lee Smith |
|Bill Smith|
|Mary Smith|
|----------|
SQL> MATCH ({name:'Peter Smith'}) [(p)-[:Child]->()]+ ({name:x})
|-----------------------------------------------------------------------|----------|
|P                                                                      |X         |
|-----------------------------------------------------------------------|----------|
|ARRAY[PERSON(ID=2,NAME=Peter Smith)]                                   |Fred Smith|
|ARRAY[PERSON(ID=2,NAME=Peter Smith),PERSON(ID=3,NAME=Mary Smith)]|Lee Smith |
|ARRAY[PERSON(ID=2,NAME=Peter Smith),PERSON(ID=3,NAME=Mary Smith)]|Bill Smith|
|ARRAY[PERSON(ID=2,NAME=Peter Smith)]                                   |Mary Smith|
|-----------------------------------------------------------------------|----------|
SQL> alter table person add primary key(name)
SQL> alter table child to childof
SQL> alter table childof alter leaving to parent
SQL> alter table childof alter arriving to child
SQL> create role ps
SQL> grant PS to
SQL>
```





```
public class CHILDOF : Versioned
{
    [Identity]
    [Field(PyrrhoDbType.Integer)]
    [AutoKey]
    public Int64? ID;
    [Leaving]
    [Field(PyrrhoDbType.String)]
    public String? PARENT;
    [Arriving]
    [Field(PyrrhoDbType.String)]
    public String? CHILD;
    0 references
    public PERSON? PARENTis => conn?.FindOne<PERSON>(("NAME", PARENT));
    1 reference
    public PERSON? CHILDis => conn?.FindOne<PERSON>(("NAME", CHILD));
}
0 references
public class Demo
{
    static PyrrhoConnect? conn = null;
    2 references
    static List<PERSON> Descendants(PERSON p)
    {
        var ds = new List<PERSON>();
        if (p.ofPARENTs is CHILDOF[] ca)
            foreach (var c in ca)
                if (c.CHILDis is PERSON d)
                {
                    ds.Add(d);
                    ds.AddRange(Descendants(d));
                }
        return ds;
    }
    0 references
    static void Main()
    {
        conn = new PyrrhoConnect("Files=ps;Role=PS");
        conn.Open();
        try
        {
            // Get a list of all descendants of Pete Smith
            var pa = conn.FindWith<PERSON>(("NAME","Peter Smith"));
            if (pa.Length == 1)
                foreach (var c in Descendants(pa[0]))
                    Console.WriteLine(c.NAME);
        }
        catch (Exception ex)
        {
```

Figure 2. A simple repeating pattern (a) Command line SQL interaction to build and display a simple database
(b) Browser display of the graph: http://localhost:8180/ps/PS/PERSON/NAME='Peter Smith'?NODE
(c) An extract from a C# client program to list Peter Smith's descendants showing PyrrhoDB's Versioned API

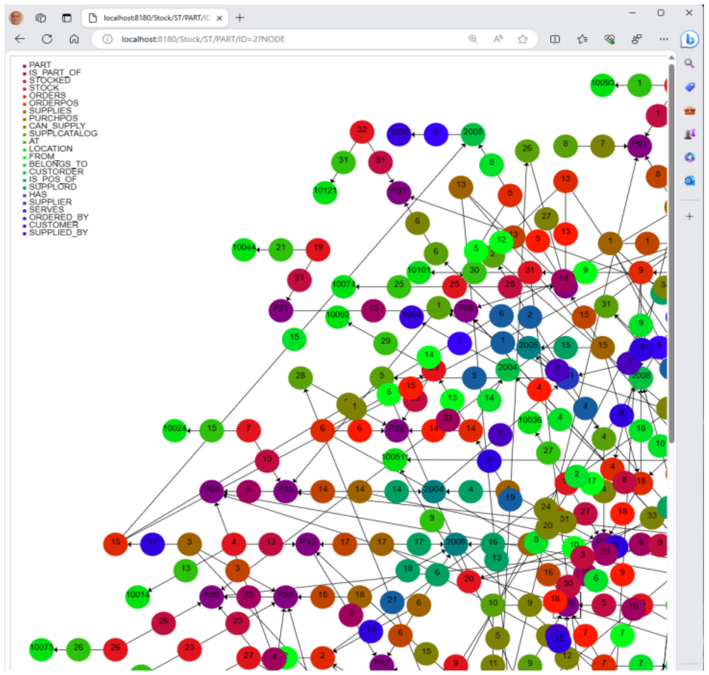


Figure 3. A part of the ERP example graph, after similar changes to primary keys (e.g., PART now has key PartID).