Rapid Exploratory Querying of Relational Data Using Hypergraphs

Joachim Klokkervoll, Samuel Nygaard,
Mike Pedersen, and Lynge Poulsgaard
{jklokk12, snpe12, mipede12, lkpo12}@student.aau.dk

May 26, 2015

Aalborg University
Department of Computer Science

Abstract

Interpretation of relational data can be complicated, as data can come from many different sources and analyzing it often requires experience with SQL. As opposed to writing queries directly in SQL, we propose a solution based on the visual representation of hypergraphs to model the data. We have designed and developed a simple query language called the Hypergraph Transform Language (HTL). HTL queries are based on the graph representation of the data and specify nodes to be removed, added, or combined in the graph. An HTL query can be directly translated into a corresponding, but more complex, SQL query that can be executed against a relational database. HTL aims at providing a more graphical conceptual understanding while being built on top of the computationally effective and widely used relational model. The hypergraph representation allows for better visualization of queries and makes it easier to identify and explore significant variables. We demonstrate these qualities by implementing a subset of HTL and providing a use case.

Aalborg, May 26, 2015

Joachim Klokkervoll
Student ID: 20124277
E-mail: jklokk12@student.aau.dk

Samuel Nygaard
Student ID: 20124278
E-mail: snpe12@student.aau.dk

Mike Pedersen
Student ID: 20124283
E-mail: mipede12@student.aau.dk

Lynge Poulsgaard
Student ID: 20124272
E-mail: lkpo12@student.aau.dk

1 Introduction

The use of databases in epidemiology is widely used [1] and, as explained by Ray [1], there exists pitfalls where researchers make erroneous conclusions based on misunderstandings, malpractice, or data linkage errors in databases. As an example, a flawed conclusion was made in 1992 where researchers concluded that women with breast implants had a significantly lower risk of breast cancer than the general population [2]. The conclusion was suggested to be caused by the researchers unfamiliarity with the different databases used in their study [1].

Management of relational databases is often associated with the commonly used Structured Query Language (SQL), which is used to insert and retrieve information. One of the main advantages of SQL is its efficiency at handling large amounts of records from a database. On the other hand, writing SQL queries may require experience and knowledge within the field of database management systems. The overall objective of this paper is to develop a tool that provides the researcher with a more graphical conceptual understanding of the data and while retaining the efficiency that SQL provides.

The tool presented in this paper enables analysis of relational data in general, and is as such not limited to using data from any specific type of application. However, as we have access to anonymized hospital records through cooperation with the Danish data-warehousing company EnVersion[1], this data will be the foundation on which the tool is tested.

Our tool is aimed at providing an easy-to-use approach to represent, manipulate and extract statistics from relational data. The tool uses the structure and properties of hypergraphs to represent and manipulate data. To ease processing and conceptual understanding, we introduce the concept of a hypergraph schema, which define classes of hyperedges and classes of nodes. To manipulate this graph representation we introduce a simple query language, the Hypergraph Transformation Language (HTL), making it easy to understand and write queries for our tool.

[1] http://enversion.dk/

The focus of the tool is to provide a conceptually understandable way of modeling and manipulating relational data as a hypergraph. As such this paper will not be concerned with the initial selection of data and the processing of the data afterwards.

1.1 Related work

The need for user-friendly data management tools has also been investigated by Horiguchi et al. [3]. They have developed a solution comparable to ours; an easy-to-use tool for research in epidemiology that abstracts from the underlying data model. The presented solution is based on a scripting language designed for user friendliness. Their solution differs from ours as it is based on the MapReduce paradigm used in the storage and processing framework Hadoop to ensure parallelism. Furthermore their tool requires the researcher to be familiar with SQL, which is not necessary with our tool.

Other similar work have been conducted in analyzing complex contextual data. A paper by Kıcıman et al. [4], concerning contextual data on the social media website Twitter, presents a framework based on hypergraphs, making it easier to inspect and analyze contextual information in the data.

2 Background

In this section, we first discuss what we believe to be one of the most used data models of today; relational data modeling. Then we discuss how data modeling as graphs is used in database systems. Finally, we briefly outline the premises for our development approach of the tool.

2.1 Relational data modeling

One way of storing and manipulating information is by using a relational data model. In the relational model the information is stored as tuples that are grouped into relations. Commonly, this model is implemented using an SQL database system. In SQL databases, these relations are defined as tables and the individual tuples within the relation are called rows. Information can then be extracted by creating a query in SQL and executing it on the database system [5]. Relational databases are a tried and tested technology and sees extensive use in many software fields.

Dimensional modeling is a special type of relational modeling in which the data is only stored in two types of relations: facts and dimensions. References and relationships occur only from fact relations to dimension relations. The dimensions that are referenced by a fact are also called the context of the fact, e.g. the patient and hospital involved in an admission fact. The fact stores the information for its context, e.g. the time of admission [6].

While most relational database design paradigms seek to reduce redundancy, dimensional modeling seeks to reduce query time at the expense of adding more redundancy. This increases insertion time, but has the possibility of speeding up querying significantly [6].

2.2 Graph databases

Modeling data visually as graphs can often give a better overview of the represented data.

A graph is a model of a set of objects and their connectivity. More formally, a graph is a tuple (V, E), where V is a set of *nodes* and E is a set of pairs of nodes (u, v) that specify an *edge* between u and v.

Graph databases are systems that store data using an underlying graph structure. Graph databases are used in cases where the queries are focused on the structure of a graph. For example, a graph database is often better at finding the shortest path between two nodes than a relational database is. This comes at a cost: graph databases tend to be significantly slower than relational databases for queries that are not dependent on the structure of the graph, such as counting the number of elements [7].

Graph database management systems are also concerned with the underlying storage system and processing engine of the database. A graph database can use a so-called native solution that is designed for storing and managing graphs with optimization in mind. However, graph data can also be serialized and stored in relational or object-oriented databases [7].

2.3 Premises

To facilitate rapid exploratory querying, we have created a tool that is both conceptually easy to understand and executes fast. The queries for our use case do not utilize the structure of the graph, so using a graph database would not yield considerable performance improvements. Ideally, we want to combine the ease of understanding from graph databases with the computational benefits of a relational database, possibly structured using dimensional modeling.

Our approach is therefore to interpret an existing relational database as a hypergraph; a graph structure that will be further explained in Section 3. The user can write queries against this virtual hypergraph, which will be translated into a corresponding SQL query, that can be executed on the underlying relational database. This design leverages the existing SQL technology, but has the potential of being conceptually simpler.

3 Hypergraphs

A hypergraph is a structure similar to standard graphs, but differs by the definition that an hyperedge may connect an arbitrary number of nodes. We define a hypergraph as a tuple (V, H), where V is a set of nodes and H is a set of multisets of nodes. Each multiset $h \in H$, $h \subseteq V$ defines a single hyperedge connecting the nodes contained in the multiset. This definition does not allow duplicate hyperedges (i.e., edges that connect the same nodes), but does allow a hyperedge to be connected to a single node multiple times (see Figure 1a for an example). As hypergraphs are a generalization of graphs, every graph is also a hypergraph.

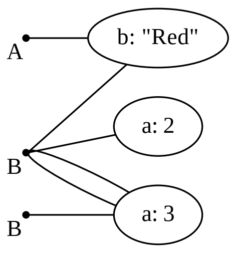
Hypergraphs provide a useful way of representing data, because it easily visualizes the data and can therefore be more comprehensible than a relational model.

3.1 Hypergraph Schema

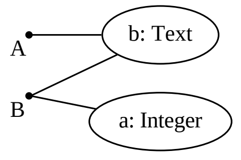
Similar to the schema for a relational database, we can define a schema for the hypergraph that describes the structure of the data. Having a schema for our data allows us to reason about the data and what effects operations on the data will have and therefore enables us to generate more efficient SQL code.

To define such a schema, we divide nodes and edges into *classes*. A node belongs to a specific class which defines its domain, e.g., whether a class of nodes contains integers or text strings. Likewise, hyperedges belong to a class defining what kind of nodes may be connected to the hyperedge. If schema S describes hypergraph G, we say that G is an *instance* of S and that G is an *instance hypergraph*.

Such a schema can itself be defined as a hypergraph. We define a *hypergraph schema* as a hypergraph where every edge in the schema defines a class of edges in the instance hypergraph. Similarly, every node in the schema



(a) Instance hypergraph



(b) Schema hypergraph

Figure 1: An example schema and hypergraph where dots are representing hyperedges

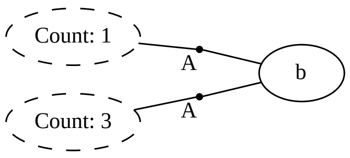
defines a class of nodes in the instance hypergraph. If hyperedge α is related to node β, then all hyperedges of class α may be related to any number of β nodes (including none).

This is demonstrated in Figure 1, where an instance hypergraph and corresponding schema is shown. The schema in Figure 1b has two edge classes **A** and **B**, and two node classes **a** and **b**. Likewise, the hypergraph in Figure 1a has one *instance* of **A**, two instances of **B**, two instances of **a**, and one instance of **b**.

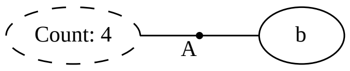
3.2 Statistics

In order to gather information over many edges, we define the concept of statistics. A statistic is a value belonging to an edge describing some quantity related to that edge. For example, a statistic might be the count of a particular class of nodes on an edge. Statistics behave similarly to nodes, except for the important difference that hyperedges which only differ in statistics are not allowed. Similar to Kıcıman et al. [4], we refer to edges that have duplicates or differ only by statistics as being *improper*. A hypergraph containing an improper edge is also improper.

Improper hypergraphs cannot be created in HTL, but can be thought of as a intermediate representation. If at some intermediate step we have an improper hyperedge, we collect the statistics of the improper hyperedge into a single proper hyperedge using the ag-



(a) Improper hypergraph



(b) Proper hypergraph

Figure 2: Aggregation of statistics, statistics shown as nodes with dashed lines.

gregate functions defined on the statistics. If the defined aggregate functions are commutative and associative, it is possible to optimize HTL on the order of execution without changing the resulting statistics. For this reason we require all aggregate functions to be both commutative and associative.

For example, we might be interested in the number of edges in a certain class. We would then initialize a statistic with the value 1 and the aggregate function sum on every edge of this class. After a projection (defined in Section 4) we might have an improper hypergraph, as seen in Figure 2a. These statistics are then aggregated using the sum function to form Figure 2b.

4 HTL language design

We implement the Hypergraph Transformation Language (HTL) as a simple language on top of SQL. HTL has been designed with inspiration from SQL and the work of Kıcıman et al. [4]. The design of HTL aims to use the conceptual representation of graphs, resulting in a better understanding of the data, while also using the computational efficiency of SQL.

Unlike SQL, it is an imperative, stateful language; querying is modeled in a sequence of steps that each modify parts of the hypergraph. Each step consists of a single statement that changes the hypergraph in some way.

4.1 Statements

HTL consist of five statements:

SELECT

```
SELECT expr AS node FOR edge
```

The selection statement adds a new class of nodes to a class of hyperedges. The name of the class of nodes is defined by the `node` parameter and must be unique for the entire hypergraph. The initial value of the nodes, defined by `expr`, can be any expression resulting in an atomic SQL value such as an integer, a text string or a function of some node or value.

AGGREGATE

```
AGGREGATE value AS statistic FOR edge
    USING aggregatefunction
```

The `AGGREGATE` statement adds an aggregation function and the associated statistic to a class of nodes. As explained in Section 3.2, this is useful in gathering information across many edges.

The initial value of the statistics is defined as the `value` parameter. As with the `SELECT` statement, the name, defined by the `statistic` parameter, must be unique for the hypergraph. Implemented aggregate functions are: min, max, and sum, but can be extended by any commutative and associative function using the HTL API. As of now the aggregate functions implemented are mostly useful for integers. However, aggregate functions for text strings could be implemented as well. An example could be a histogram, also known as term frequency vector, aggregate counting the appearance of characters or words.

PROJECT

```
PROJECT [ AWAY | TO ] nodes
```

Some data might be irrelevant or more interesting as statistics than actual values. In that case one could remove the nodes completely from the hypergraph or aggregate over them. For this purpose the PROJECT statement is introduced.

Using projection with the AWAY keyword removes one or more classes of nodes, given by the comma-separated list of nodes in the nodes parameter. If nodes are projected away, edges might become improper in the sense that one edge is of the same class as another and is connected to the same nodes. If any such edges exist, the edges are combined into a single edge, aggregating any statistics onto the resulting hyperedge, as defined by the associated aggregate functions. Statistics will continue to be aggregated if edges are combined.

Using projection with the TO keyword removes all other classes of nodes than those specified in the nodes parameter. As with PROJECT AWAY statistics are aggregated according to the associated aggregate functions.

COMBINE

```
COMBINE nodes AS node
```

Data from different sources might share specific values, e.g. the ID of a person, combining these nodes can be useful in revealing connections in the data. This statement communicates to the system that two nodes are semantically the same.

Combine merges two or more classes of nodes, as defined by the nodes parameter, into a single new class of nodes defined by the node parameter. Nodes with the same values are merged into a single node. This can be used to connect hyperedge classes that are not directly connected.

An example could be two distinct classes of hyperedges where one is connected to a person_id node and the other a patient_id node. These two classes of edges are not connected, however the person_id and patient_id nodes might in fact be referring to the same actual id, meaning that it would make sense to combine these two classes of nodes into one.

RELATE BY

```
RELATE BY nodes AS edge
```

Revealing connections in data can provide interesting results when performing data analysis. Relations can be interpreted for data sharing semantically equal values. These relationships can be imposed onto the hypergraph by using a RELATE BY statement.

RELATE BY merges any hyperedges that share a common node of any of the classes defined by the nodes parameter. When edges are merged, any statistics will be aggregated to the resulting hyperedges using the associated aggregate function. RELATE BY defines a new class of edges containing all merged hyperedges. A unique edge name for this class must be provided in the edge parameter.

Order of statements

The statements can appear in any order. The order might, however, affect performance. Especially projections might speed up the execution of a query. If projection statements are put earlier in the ordering, the working set can be severely reduced, reducing the time used for following statements. The order of statements might also affect the resulting graph, an example can be the order of the following statements:

```
SELECT 1 AS tempNode FOR tempEdge
PROJECT TO otherNode;
```

Code Listing 1 The HTL query that answers **Q**.

```
AGGREGATE 1 AS count FOR stay USING
    SUM
PROJECT AWAY stay.patient_id
COMBINE stay.id, stay_diagnose.id AS
    stay_id
RELATE BY stay_id AS
    department_diagnoses
PROJECT AWAY stay_id;
```

If the statements are executed in the given order the `tempNode` will not be part of the resulting hypergraph. If, however, the `PROJECT` statement is placed first the `tempNode` will be part of the result.

4.2 Example

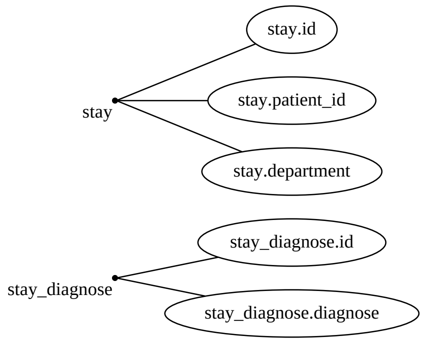
As an example of how to use HTL for querying and to illustrate the simplicity of using HTL, consider the following query for a hospital database:

Q At what frequencies do different hospital departments register different combinations of diagnoses?

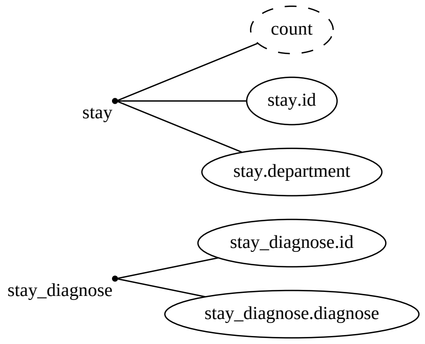
Initially the graph might contain hyperedges for hospital stays (called `stay`) and their diagnoses (called `stay_diagnose`). An example of such a schema can be seen in Figure 3a.

The HTL query that will produce the result of **Q** on such a schema can be seen in Listing 1. This query is semantically equivalent to the handwritten PostgreSQL query seen in Listing 2.

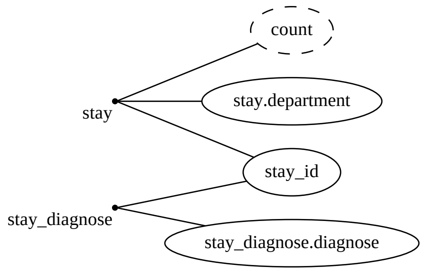
The resulting hypergraph schema and intermediate steps, produced when running the HTL query in Listing 1, is seen in Figure 3. The final schema would contain a hyperedge for every hospital department, as seen in Figure 3d. Each hyperedge connects a department to combinations of diagnoses and contains a statistic that tells how many times the department has seen this specific combination of diagnoses.



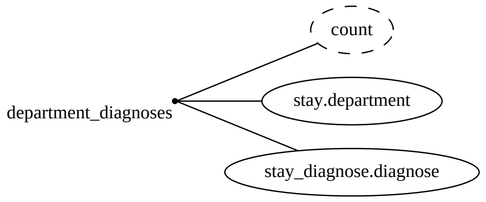
(a) Initial relational layout.



(b) `count` has been added as a statistic. `stay.patient_id` have been projected away.



(c) `stay.id` and `stay_diagnose.id` have been combined into `stay_id`.



(d) Relating by `stay_id` and projecting the node away reveals the connection between department and diagnoses.

Figure 3: The hypergraph transformation steps made when executing **Q**.

Code Listing 2 The equivalent SQL query for **Q**. Note that both a nested query and aggregation method is required.

```
WITH stay_id AS (
  SELECT department,
      ARRAY_AGG(DISTINCT diagnose
      ORDER BY diagnose) AS diagnose
  FROM stay a JOIN stay_diagnose b ON
      (a.id = b.id)
  GROUP BY a.id, department)
SELECT department, diagnose, COUNT(*)
FROM stay_id
GROUP BY department, diagnose;
```

As the example shows, it is clear to see that the HTL queries takes a simpler form than their SQL equivalent queries. The SQL query would grow even larger when using several `RELATE BY` statements in the HTL query.

The HTL query in Listing 1 uses the projection statement on the `stay_id` node which have been used in a `RELATE BY` earlier. The resulting SQL of Listing 2 omitting the final projection can be seen in Appendix A. The reason for the omission of `PROJECT` is discussed in Section 5.2.

5 Implementation

Our application interprets an SQL-based relational database as a hypergraph by making each table into a class of hyperedges and each column into a class of nodes. An example of a hypergraph schema and the corresponding relational schema can be seen in Figure 4. Initially, there will be a one-to-one correspondence between hyperedges in the graph and rows in the database. All operations except `RELATE BY` keep this invariant (further explained in Section 5.1).

This representation of the hypergraph can be queried by writing a query in HTL. Each statement in HTL is translated to SQL queries which are then combined as subqueries using

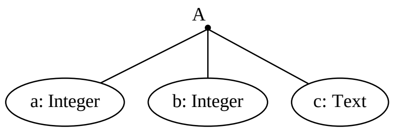


Figure 4: An example hypergraph schema, equivalent to the relational schema A(a integer, b integer, c text)

Code Listing 3 Sample HTL query

```
AGGREGATE 1 AS c FOR a USING SUM
PROJECT TO b;
```

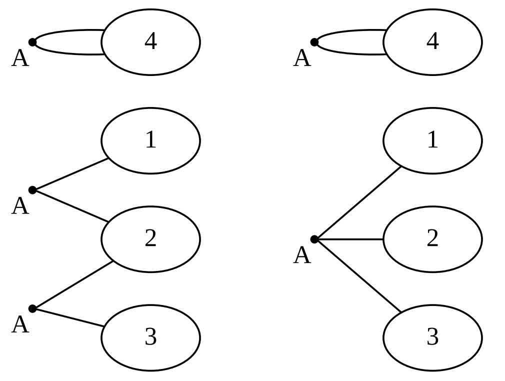
SQL's `WITH` clause. The result is a single SQL query for the entire HTL query that can be executed on a relational database system.

Most of the HTL operations map cleanly to SQL; `SELECT` and `AGGREGATE` can be accomplished using SQL's `SELECT` and aggregate functions. Aggregate functions can be any SQL expression, as long as it return a single atomic value and is both commutative and associative. `PROJECT` can be done using SQL's `GROUP BY` clause by grouping by all nodes and using aggregate functions to collect statistics. An example of this can be seen in Listing 3, where an HTL query generates the SQL query seen in Listing 4.

Code Listing 4 Resulting SQL query of Listing 3

```
WITH t1 AS (SELECT *, 1 AS c FROM a),
     t2 AS (SELECT b, SUM(c) AS c
         FROM t1 GROUP BY b)
SELECT * FROM t2;
```

The generated SQL conforms to the ISO/IEC SQL:1999 standard[8], with the only exception being our use of the common, but nonstandard, `LEAST` and `GREATEST` functions. `LEAST` and `GREATEST` are functions that return the minimum and maximum value of its parameters. The functionality of these functions can be recreated using SQL's `CASE` statements, but the complexity of the resulting SQL is significantly higher.



(a) Before `RELATE BY` **(b)** After `RELATE BY`

Figure 5: Example of `RELATE BY` applied to an instance hypergraph with two connected components

We have only found PostgreSQL to fully support the features we utilize, such as recursive common table expressions together with the `LEAST` and `GREATEST` functions.

5.1 Implementation of `RELATE BY`

The most complex operation is the `RELATE BY` operation. The operation takes a set of node classes (hereafter called the relate set) and combines all hyperedges sharing a node in that set. This problem reduces to finding *connected components* in the hypergraph, considering only the nodes in the relate set. A connected component is a subgraph wherein there is a path from any node to any other node. In our case this would be edges that are connected through the nodes specified in the relate set. An example of finding connected components in a graph can be seen in Figure 5.

As previously noted, the `RELATE BY` statement does not keep a one-to-one correspondence between hyperedges and rows. This is a necessity, due to `RELATE BY` connecting an unpredictable number of nodes to a single hyperedge. In the worst case, the entire hypergraph may form a single connected component. In the best case, each edge is its own connected component. If the result of a `RELATE BY` operation were to be represented as a single row, it would require a dynamic number of columns which is not possible in standard SQL.

a	**b**
1	2
2	3
4	4

(a) Before `RELATE BY`

Comp	**a**	**b**
1	1	2
1	2	3
4	4	4

(b) After `RELATE BY`

Table 1: A relational model corresponding to Figure 5

Implementing `RELATE BY` efficiently in SQL is problematic, but doable. We augment the table with an additional column storing a component index, based on the columns in the relate set. For example, consider Table 1 where a simple relational table corresponding to Figure 5 is shown. A `RELATE BY` on Table 1a will augment the table with a new column **Comp** as seen in Table 1b. Note that the operation will identify two connected components, which will correspond to two hyperedges in the graph even though this is three rows in the relational representation.

The algorithm itself is implemented using a recursive SQL common table expression. Assuming no major optimization by the compiler, the worst case expected running time will be at least $o(nc)$ where n is the number of rows and c is the number of rows in the largest component. This potentially quadratic running time (or worse, depending on how the database implements it) can make the implementation unusable in cases where large components are created. In many cases this can make the `RELATE BY` operator impractical.

We can, however, utilize our knowledge of the underlying relational representation to lower the algorithmic complexity of `RELATE BY`. Consider if we relate over primary keys and foreign keys to those primary keys, then we can replace the entire recursive `RELATE BY` with a single SQL `SELECT` statement since we can deduce the component index from the foreign key.

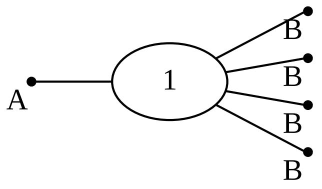


Figure 6: Example of a star-shaped instance hypergraph.

Comp	a
1	2
2	2

(a) Improper hypergraph

Comp	a
1	2
2	2
2	3

(b) Proper hypergraph

Table 2: Example of the problem of identifying improper nodes

This pattern can be visualized as a star pattern in the instance hypergraph, as seen in Figure 6 and occurs often for relational data. This optimization can improve speed significantly to the point where `RELATE BY` is practical. This optimization, while important for `RELATE BY`, is currently not implemented in our tool. Implementing this optimization should be a priority for future versions.

5.2 Limitations

In the current implementation of HTL there exists some limitations. A very limiting issue with the implementation of `RELATE BY` is the inability to correctly aggregate statistics when using `PROJECT AWAY` on the node that was related by. The problem is, that when hyperedges are represented as several rows, it becomes non-trivial to check for improper edges, i.e. edges that should be merged, in linear time. This problem is significant, as many uses of HTL depends on being able to `PROJECT AWAY` the nodes that was used in the `RELATE BY` clause.

An example of why this problem occurs can be seen in Table 2. Table 2a is improper because there are two edges (identified by component index 1 and 2) that are equivalent, while Table 2b is proper due to component 2 having one more connection. The problem arises due to the difficulty of disambiguating these two cases in SQL and aggregating statistics correctly.

At this time we do not know if there exists a viable solution to this limitation. In our case study, we manually transformed the HTL query into SQL just like an implementation of HTL without these limitations would. This allowed us to avoid the limitations of the current implementation for our case study.

6 Case study

To assess the features of the hypergraph model we present a concrete case for application. It consists of analyzing the diagnoses of patients and we formulate it as follows: *what diagnoses tend to lead to other diagnoses?* In answering this question we might see that patients diagnosed with **A** often get diagnosed with **B** later in the same treatment. This might be because **B** is a sequela of **A**, because **B** was initially misdiagnosed as **A**, or simply a coincidence. The core of this problem is to find how closely related different diagnoses are.

For this case study we base our analysis on the data set that we have received in collaboration with the Danish data warehousing company EnVersion and the Regions of Denmark[2]. The data is structured using dimensional modeling (see Section 2.1), where the facts consist of timestamped events from the hospitals, e.g. admissions and procedures. The dimensions consists of non-timestamped data such as procedure codes, drug doses, and patient information.

The initial interpretation of the hypergraph contains nodes that we are not interested in. We start by projecting these away. The result of this projection can be seen in Figure 7a.

[2]Five regional areas of Denmark that each administrates the social services of its citizens

Code Listing 5 HTL query for related diagnoses

```
PROJECT TO
    intervention_diagnose.diagnose,
    intervention_diagnose.id,
    intervention.id,
    intervention.course, stay.course,
    stay.id, stay_diagnose.id,
    stay_diagnose.diagnose
COMBINE intervention_diagnose.id,
    intervention.id AS
    intervention_diag
COMBINE stay.id, stay_diagnose.id AS
    stay_diag
COMBINE intervention.course,
    stay.course AS course
AGGREGATE 1 AS count FOR stay USING
    SUM
RELATE BY intervention_diag,
    stay_diag, course
PROJECT AWAY intervention_diag,
    stay_diag, course;
```

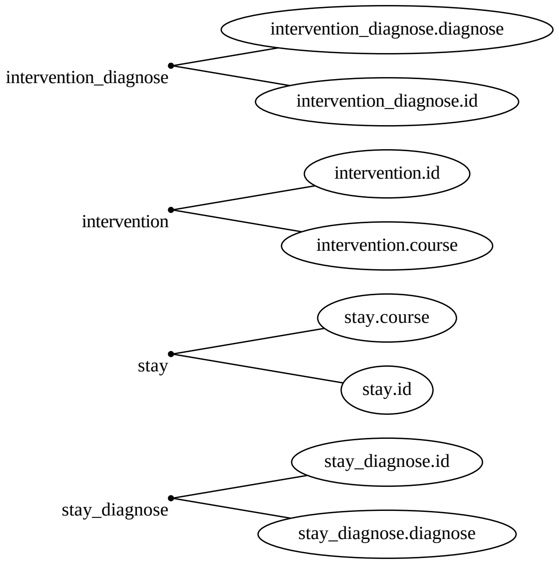
Afterwards we want to combine ID and course nodes, since these are semantically identical. The result of this COMBINE is then seen in Figure 7b.

We want to count the number of times one diagnose relates to another. We therefore apply a statistic count onto stay with the initial value 1 and aggregate function SUM.

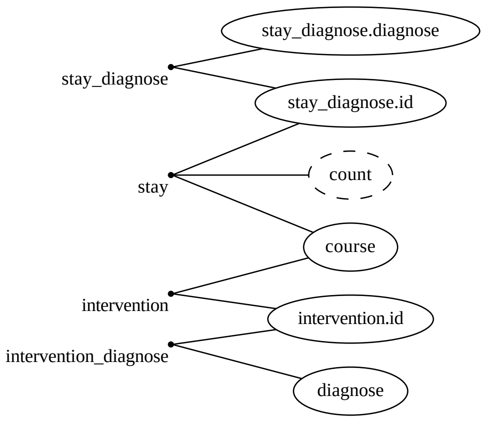
Since we are interested in how stay diagnoses are related to intervention diagnoses, we want to relate by all the intermediate edges. After this RELATE BY, we have the final hypergraph schema seen in Figure 7c.

The HTL query that expresses these steps can be seen in Listing 5. One can then utilize a number of different machine learning tools to analyze the resulting hypergraph. One could, for example, interpret the count as an inverse distance and use a clustering algorithm to find clusters of diagnoses that often occur together.

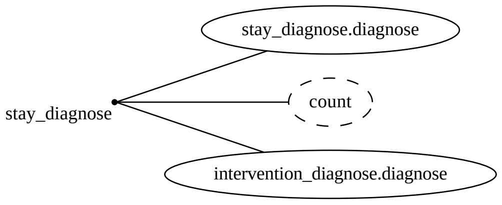
There may be additional variables that is left unexplored. If these variables are connected to our initial edges, it is simply a case of not projecting these nodes away. If we wanted to know how the gender of patients affects the statistics



(a) After initial `PROJECT`



(b) After `COMBINE`



(c) After `RELATE BY`

Figure 7: Hypergraph schema of the use case in different stages

aggregated over the diagnoses we would just append gender to the initial PROJECT TO like so:

```
PROJECT TO
    intervention_diagnose.diagnose,
    ... , gender
```

This use case therefore demonstrates that HTL makes it easy to quickly discover what variables might be significant. In Appendix B.1 the generated SQL query of Listing 5 omitting the final projection is shown. An SQL programmer would likely produce an PostgreSQL query similar to the one seen in Appendix B.2 which correctly generates the result that our HTL query asks for.

7 Discussion

Our case study shows the benefits of using HTL in finding connections in relational data. The example in Section 4.2 has shown how complex SQL queries can be expressed more easily using HTL. Regarding the usability of HTL versus SQL, having only five simple statements with greater orthogonality might make HTL more clear and easy to use for a researcher without prior knowledge of query languages. However, the improved ease of expression could reduce performance as the generated SQL might not be fully optimizable.

There are a number of operations currently not available in HTL that would be useful. Filtering, for example, is a very useful operation that currently cannot be expressed in any way. One way to address this deficiency would be to allow the SELECT clause to only add nodes from the class meeting a certain criteria or constraint on their value. Another solution would be to extend the PROJECT TO statement with an optional constraint predicate such as PROJECT TO nodeclass WHERE VALUE > 2, which would make HTL filter away irrelevant nodes. Of course, data filtering can be done beforehand at the database level by creating views or temporary tables containing only the filtered data. However, as HTL strives to eliminate the needs of interacting with a database using SQL queries, supporting data filtering in HTL would be a good choice of focus for future version.

As explained in Section 4, the order of operations plays a large role in the efficiency of a query. Some of these operations are order-independent and can thus be rearranged while having no impact on the results. An optimization phase for this would remove the need for manually doing this optimization and would be a good feature for a future version.

The complications in finding connected components is an unanswered issue as we have yet to find a good solution in SQL. Finding connected components is in itself not a difficult task, but within the framework of SQL it becomes computationally challenging, as discussed in Section 5.2.

A better solution might be to use a disjoint-set structure in a general purpose programming language. Disjoint-set forests are very efficient data structures that allows two operations: finding a representative element and creating the union of two subsets [9]. Both of these can be done in practically constant time [9], which would enable us to find all connected components in linear time, instead of the potentially quadratic running time of the current implementation.

This would lose some of the immediate benefits compiling to SQL would have. SQL can do the computation directly on the database, whereas using a more general purpose programming language would likely require extracting the data and storing it as some intermediate form.

8 Conclusion

In trying to create a conceptually more understandable way of querying relational data, we designed and built a querying tool using hypergraphs to represent the data. The implementation of the tool introduced the HTL language and the hypergraph schema, both proved to be useful in querying relational data. Our case study showed that the inception of HTL as a tool for querying relational data as a graph representation reduces the fundamental understanding required to effectively analyze and find patterns in relational data. Since HTL compiles to SQL, it retains the efficiency of data manipulation that SQL entails. As mentioned in Section 5.2, the current implementation has problems with projecting away a class of nodes that was used to create a relation between hyperedges. A possible future implementation would primarily be focused on removing these limitations as well as implementing the additional features as mentioned in Section 7.

References

[1] Wayne A. Ray. Improving automated database studies. *Epidemiology*, 22(3):302–304, 2011.

[2] Heather Bryant and Penny Brasher. Breast implants and breast cancer - reanalysis of a linkage study. *New England Journal of Medicine*, 332(23):1535–1539, 1995.

[3] Hiromasa Horiguchi, Hideo Yasunaga, Hideki Hashimoto, and Kazuhiko Ohe. A user-friendly tool to transform large scale administrative data into wide table format using a mapreduce program with a pig latin based script. *BMC medical informatics and decision making*, 12(1):151, 2012.

[4] Emre Kıcıman, Scott Counts, Michael Gamon, Munmun De Choudhury, and Bo Thiesson. Discussion graphs: Putting social media analysis in context. In *Eighth International AAAI Conference on Weblogs and Social Media*, 2014.

[5] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Science/Engineering/-Math, 2010. ISBN 0073523321.

[6] Ralph Kimball. *The data warehouse lifecycle toolkit*. Wiley Pub, Indianapolis, IN, 2008. ISBN 9780470149775.

[7] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases*. " O'Reilly Media, Inc.", 2013.

[8] ISO. Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation), 12 1999.

[9] Thomas Cormen. *Introduction to Algorithms*. MIT Press, Cambridge, Mass, 2001. ISBN 0262032937.

APPENDIX

This section includes the SQL code generated from the HTL queries used in the example in Section 4.2 and the case study in Section 6. Both HTL queries use a PROJECT statement on a node that was related by. As explained in Section 5.2 this is not supported in the current implementation. As such the last PROJECT statement of both queries have been left out for the generated code below.

A Example query

```
WITH tbl0
AS (
  SELECT *, 1 AS sel1
  FROM stay),
tbl1 AS (
  SELECT id, department, (SUM(sel1)) AS sel1
  FROM tbl0
  GROUP BY id, department ),
tbl2 AS (
  SELECT *
  FROM tbl1 ),
tbl3 AS (
  SELECT id, diagnose
  FROM stay_diagnose
  GROUP BY id, diagnose ),
tbl4 AS (
  SELECT *
  FROM tbl3 ),
tbl5 AS (
  SELECT id AS f0, department AS f1, sel1 AS f2, NULL AS f3, NULL AS f4
  FROM tbl2

  UNION ALL
  (
    SELECT NULL AS f0, NULL AS f1, NULL AS f2, id AS f3,diagnose AS f4
    FROM tbl4 )
  ),
tbl6 AS (
  WITH RECURSIVE "@cl" (
      comp0, f0, f1, f2, f3, f4 )
      AS (
      SELECT LEAST(f0, f3), tbl5.f0, tbl5.f1, tbl5.f2, tbl5.f3, tbl5.f4
      FROM tbl5

      UNION

      SELECT "@cl".comp0, tbl5.f0, tbl5.f1, tbl5.f2, tbl5.f3, tbl5.f4
      FROM "@cl"
      INNER JOIN tbl5 ON "@cl".f0 = tbl5.f0 OR "@cl".f0 = tbl5.f3 OR "@cl".f3
         = tbl5.f0 OR "@cl".f3 = tbl5.f3
      )
  SELECT min(comp0) AS comp0, f0, f1, f2, f3, f4
  FROM "@cl"
  GROUP BY f0, f1, f2, f3, f4
  )
SELECT f0, f1, f3, f4, comp0, (SUM(f2)) AS f2
FROM tbl6
GROUP BY f0, f1, f3, f4, comp0
ORDER BY comp0
```

B Case study queries

B.1 Generated SQL query

```
WITH tbl0 AS (
  SELECT id, course
  FROM intervention
  GROUP BY id, course ),
tbl1 AS (
  SELECT *
  FROM tbl0 ),
tbl2 AS (
  SELECT *
  FROM tbl1 ),
tbl3 AS (
  SELECT id, course
  FROM stay
  GROUP BY id, course ),
tbl4 AS (
  SELECT *
  FROM tbl3 ),
tbl5 AS (
  SELECT *
  FROM tbl4 ),
tbl6 AS (
  SELECT *, 1 AS sel1
  FROM tbl5 ),
tbl7 AS (
  SELECT id, diagnose
  FROM stay_diagnose
  GROUP BY id ,diagnose ),
tbl8 AS (
  SELECT *
  FROM tbl7 ),
tbl9 AS (
  SELECT id, diagnose
  FROM intervention_diagnose
  GROUP BY id, diagnose ),
tbl10 AS (
  SELECT *
  FROM tbl9 ),
tbl11 AS (
    SELECT id AS f0, course AS f1, NULL AS f2, NULL AS f3, NULL AS f4, NULL AS
        f5, NULL AS f6, NULL AS f7, NULL AS f8
    FROM tbl2
    UNION ALL
    (
      SELECT NULL AS f0, NULL AS f1, id AS f2, course AS f3, sel1 AS f4, NULL
          AS f5, NULL AS f6, NULL AS f7, NULL AS f8
      FROM tbl6
    )
    UNION ALL
    (
      SELECT NULL AS f0, NULL AS f1, NULL AS f2, NULL AS f3, NULL AS f4, id AS
          f5, diagnose AS f6, NULL AS f7, NULL AS f8
      FROM tbl8
    )
    UNION ALL
    (
      SELECT NULL AS f0, NULL AS f1, NULL AS f2, NULL AS f3, NULL AS f4, NULL
          AS f5, NULL AS f6, id AS f7, diagnose AS f8
      FROM tbl10
    )
),
tbl12 AS (
  WITH RECURSIVE "@cl" (
      comp2, comp1, comp0, f0, f1, f2, f3, f4, f5, f6, f7, f8
  )
  AS (
      SELECT LEAST(f0, f7), LEAST(f1, f3), LEAST(f2, f5), tbl11.f0, tbl11.f1,
          tbl11.f2, tbl11.f3, tbl11.f4, tbl11.f5, tbl11.f6, tbl11.f7, tbl11.f8
      FROM tbl11

      UNION

      SELECT "@cl".comp2, "@cl".comp1, "@cl".comp0, tbl11.f0, tbl11.f1,
          tbl11.f2, tbl11.f3, tbl11.f4, tbl11.f5, tbl11.f6, tbl11.f7, tbl11.f8
      FROM "@cl"

      INNER JOIN tbl11 ON "@cl".f2 = tbl11.f2 OR "@cl".f2 = tbl11.f5 OR
          "@cl".f5 = tbl11.f2 OR "@cl".f5 = tbl11.f5 OR "@cl".f1 = tbl11.f1 OR
          "@cl".f1 = tbl11.f3 OR "@cl".f3 = tbl11.f1 OR "@cl".f3 = tbl11.f3 OR
          "@cl".f0 = tbl11.f0 OR "@cl".f0 = tbl11.f7 OR "@cl".f7 = tbl11.f0 OR
          "@cl".f7 = tbl11.f7
      )
  SELECT min(comp2) AS comp2, min(comp1) AS comp1,
        min(comp0) AS comp0, f0, f1, f2, f3, f4, f5, f6, f7, f8
  FROM "@cl"
  GROUP BY f0, f1, f2, f3, f4, f5, f6, f7, f8
),
tbl13 AS (
  SELECT f6, f8, (SUM(f4)) AS f4
  FROM tbl12
  GROUP BY f6,f8
  )
SELECT f6, f8, ((SUM(f4))) AS f4
FROM tbl13
GROUP BY f6,f8;
```

B.2 Handwritten SQL query

```
WITH stay_relateby AS (
  SELECT course, array_agg(DISTINCT diagnose ORDER BY diagnose) stay_diag
  FROM stay_diagnose d JOIN stay s ON (d.id = s.id)
  GROUP BY course),

   intervention_relateby AS (
  SELECT course, array_agg(DISTINCT diagnose ORDER BY diagnose)
     intervention_diag
  FROM intervention_diagnose d JOIN intervention i ON (d.id = i.id)
  GROUP BY course)
SELECT stay_diag, intervention_diag, COUNT(*)
FROM stay_relateby NATURAL JOIN intervention_relateby
GROUP BY stay_diag, intervention_diag;
```