HKU CS Tech Report TR-2013-02

Automating Relational Database Schema Design for Very Large Semantic Datasets

Thomas Y. Lee, David W. Cheung, Jimmy Chiu, S.D. Lee, Hailey Zhu, Patrick Yee, Wenjun Yuan

Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong

Abstract

Many semantic datasets or RDF datasets are very large but have no pre-defined data structures. Triple stores are commonly used as RDF databases yet they cannot achieve good query performance for large datasets owing to excessive self-joins. Recent research work proposed to store RDF data in column-based databases. Yet, some study has shown that such an approach is not scalable to the number of predicates. The third common approach is to organize an RDF data set in different tables in a relational database. Multiple "correlated" predicates are maintained in the same table called *property table* so that table-joins are not needed for queries that involve only the predicates within the table. The main challenge for the property table approach is that it is infeasible to manually design good schemas for the property tables of a very large RDF dataset. We propose a novel data-mining technique called *Attribute Clustering by Table Load* (ACTL) that clusters a given set of attributes into correlated groups, so as to automatically generate the property table schemas. While ACTL is an NP-complete problem, we propose an agglomerative clustering algorithm with several effective pruning techniques to approximate the optimal solution. Experiments show that our algorithm can efficiently mine huge datasets (e.g., Wikipedia Infobox data) to generate good property table schemas, with which queries generally run faster than with triple stores and column-based databases.

Keywords: RDF database schema design, attribute clustering, Wikipedia Infobox, semantic web

1. Introduction

In conventional information systems, data are usually very structured and can be organized neatly in pre-designed tables. Simply speaking, a database consists of a set of transactions. A transaction has a set of attributes and an attribute can be associated with a value. Generally, transactions that represent the same kind of objects or concepts are regarded as in the same class. For example, teacher and course data are two different classes of transactions. Transactions in the same class tend to share the same set of attributes, e.g., *code* and *teacher* attributes for *course* transactions. Considering the attributes for the same transaction class being "correlated," a database designer seeks to group them in the same table. Since a query typically involves the attributes in only one or a few classes, storing correlated attributes in the same table can often avoid excessive table-joins. To design a good database schema, the designer must have a priori domain knowledge on the structure of the data and the pattern of the queries. However, many semantic web datasets have neither well-defined data structures nor well-known query patterns. Some datasets are so huge that it is not feasible to manually design a good database schema by analyzing the data patterns. This makes conventional database design methodologies not applicable in these cases.

Semantic web data are commonly represented by RDF[1] triplets, each comprising a *subject*, a *predicate*, and an *object*, i.e., (s p o). Although storing RDF data in a single 3-column table, a.k.a. *triple store*, is straightforward and flexible, this structure is not optimized for query processing. A typical query involving multiple predicates requires expensive self-joins of the triplet table. This makes query

Email addresses: ytlee@cs.hku.hk (Thomas Y. Lee), dcheung@cs.hku.hk (David W. Cheung), khchiu@cs.hku.hk (Jimmy Chiu), sdlee@cs.hku.hk (S.D. Lee), hyzhu@cs.hku.hk (Hailey Zhu), kcyee@cs.hku.hk (Patrick Yee), wjyuan@cs.hku.hk (Wenjun Yuan)

processing on a huge RDF dataset slow. To address this performance problem, the *vertical* or *column-based* approach was introduced.[2] This approach splits the large triplet table into many smaller 2-column tables, each storing the subjects and objects only for a single predicate. This way, self-joins of a large 3-column table are translated into table-joins between multiple small 2-column tables. However, some study[3] has shown that a column-based database sometimes performs even slower than a triple store when the number of predicates is large.

To avoid expensive table joins, *property tables* can be used.[4] In this approach, a database is organized as different property tables, each storing the objects for a distinct group of correlated predicates. Suppose p_1, p_2, p_3 are correlated predicates maintained in property table C. C has four columns, one for the subjects, and the others for the objects of p_1, p_2, p_3. If an query involves only the predicates among p_1, p_2, p_3, no table-join is needed. To store the triplets $(s\ p_1\ o_1)$ and $(s\ p_3\ o_3)$, a row is created in C where the subject cell stores s, the p_1 and p_3 cells store o_1 and o_3 respectively while the p_2 cell is null. In general, subjects semantically in the same class (e.g., course) often share the same predicates (e.g., course code and teacher). However, not many RDF datasets contain the class information. Therefore, it is crucial to have an effective automated schema design technique in order to make the property table approach useful for huge loosely structured RDF datasets.

1.1. Our Contributions

In this paper, we propose a new clustering problem called *Attribute Clustering by Table Load (ACTL)* to automate schema design of property tables. Intuitively, given a set of predicates, which we call *attributes* in this paper, ACTL aims to cluster them into disjoint clusters of attributes; each attribute cluster is used to create a property table. Intuitively, if the attributes maintained in table C are highly correlated, then the subjects, which we call *transactions* in this paper, stored in the table should share *most* of the attributes maintained in C. Hence, most of the cells in the table should be non-null. Conversely, when a large portion of the cells in C are null, we know that the correlation of some attributes is low. Therefore, C should be further split into sub-tables so that each maintains only the attributes with sufficient degree of correlation. We use the proportion of non-null cells in a table, which we call *load factor*, to measure the degree of correlation between the predicates. ACTL seeks to group correlated attributes together into as few tables as possible yet the load factors of all tables are high enough. On the one hand, the fewer tables there are, the "wider" the tables become, the less likely table-joins are needed. On the other hand, the higher the load factors are, the higher attribute correlation the tables have, the higher chance there is for queries to be answered with fewer joins. In addition, a higher load factor implies a higher storage efficiency for a table.

Recognizing the ACTL problem is NP-complete, we have developed an agglomerative clustering algorithm with pruning to approximate the optimal solution. We conducted experiments with huge real-life datasets, Wikipedia Infobox data[5] and Barton Libraries data[6]. Experimental results have demonstrated the following. Firstly, our algorithm can efficiently generate "good" schemas for these datasets. Secondly, the performance of running some common queries on the Wikipedia dataset using the property tables automatically designed by our technique is generally higher than that using the triple store and the vertical database.

1.2. Organization of This Paper

This paper is organized as follows. Sect. 1 introduces the motivation of our research. Sect. 2 compares four popular RDF storage schemes and reviews the related work. Sect. 3 formulates the ACTL problem. Sect. 4 presents a basic version of our agglomerative clustering algorithm and shows that this version of algorithm has high time and space complexity. To cope with the high complexity, we propose three pruning techniques to enhance the performance of the algorithm in Sect. 5. Sect. 6 discusses the concept of *attribute connectivity*, which is used to prevent uncorrelated attributes from being put in the same cluster. Sect. 7 introduces two metrics to measure the "fitness" of the schema of property tables against their actual, storage. Sect. 8 compares our approach to other related approaches. Sect. 9 analyzes the results of three experiments. Lastly, Sect. 10 sums up our work.

2. Preliminaries

An RDF file is essentially a collection of triplets. Each triplet is in the form of "*subject predicate object .*"[7] The subject is a resource represented by its URI. The predicate defines a property of the subject and is represented by a URI. The object is the

subject	*predicate*	*object* .
⟨Tom⟩	⟨degree⟩	“PhD” .
⟨May⟩	⟨degree⟩	“MPhil” .
⟨Roy⟩	⟨degree⟩	“BSc” .
⟨May⟩	⟨enrolls⟩	⟨Db⟩ .
⟨Roy⟩	⟨enrolls⟩	⟨Db⟩ .
⟨Roy⟩	⟨enrolls⟩	⟨Web⟩ .
⟨Sam⟩	⟨title⟩	“Professor” .
⟨Tom⟩	⟨title⟩	“Instructor” .
⟨Kat⟩	⟨title⟩	“Professor” .
⟨Tom⟩	⟨supervisor⟩	⟨Sam⟩ .
⟨May⟩	⟨supervisor⟩	⟨Sam⟩ .
⟨Sam⟩	⟨interest⟩	“Datamining” .
⟨Sam⟩	⟨interest⟩	“Database” .
⟨Kat⟩	⟨interest⟩	“Security” .
⟨Db⟩	⟨teacher⟩	⟨Sam⟩ .
⟨Net⟩	⟨teacher⟩	⟨Sam⟩ .
⟨Web⟩	⟨teacher⟩	⟨Tom⟩ .
⟨Db⟩	⟨code⟩	“C123” .
⟨Net⟩	⟨code⟩	“C246” .
⟨Web⟩	⟨code⟩	“C135” .

Figure 1: Motivating RDF example (`triples`)

```
SELECT  ?s FROM <triples>
WHERE { ?s <degree> "MPhil" .
        ?s <enrolls> <Db> .
        ?s <supervisor> <Sam> . }
```

Figure 2: SPARQL query example involving three predicates

```
SELECT a.subject
FROM triples a, triples b, triples c
WHERE a.subject = b.subject AND b.subject = c.subject
  AND a.predicate = "<degree>" AND a.object = "MPhil"
  AND a.predicate = "<enrolls>" AND a.object = "<Db>"
  AND a.predicate = "<supervisor>" AND a.object = "<Sam>";
```

Figure 3: SQL statement translated from Fig. 2 for triple store

value of the predicate, which can be a resource or a *literal* (constant value). For example:

```
<http://ex.edu/Db> <http://ex.edu/teacher>
  <http://ex.edu/Sam> .

<http://ex.edu/Sam> <http://ex.edu/title> "Professor" .

<http://ex.edu/Tom> <http://ex.edu/supervisor>
  <http://ex.edu/Sam> .
```

Fig. 1 shows an RDF file which serves as the motivating example in this paper. For simplicity, the full URIs of resources and predicates are abbreviated and angle-bracketed, while literals are double-quoted. In the following, we review four mainstream RDF storage schemes, namely *triple store*, *horizontal database*, *vertical database*, and *property tables*, and study their pros and cons.

2.1. Triple Store

The triple store approach[8] stores RDF triplets in a 3-column table. Fig. 1 can be considered as a triple store. A variant[9] of this scheme uses a symbol table to represent each different resource, predicate or literal by a unique system ID. Despite the flexibility of this scheme, it is commonly recognized to be slow in processing queries[2] for large datasets. It is because a query would likely require many self-joins of a huge triplet table. For instance, the SPARQL[10] query shown in Fig. 2 is translated into the SQL statement for the triple store shown in Fig. 3, which requires two self-joins.

2.2. Horizontal Database

The horizontal database approach[11] uses a single “universal table” where each row represents a subject and each column stores the objects of a distinct predicate. Note that a cell may store multiple values because of the multi-valued nature of RDF properties. Fig. 4 shows the content of the horizontal database for the motivating example. This scheme can save table-joins for a query involving multiple predicates. For example, the SPARQL query in Fig. 2 is translated into the SQL query in Fig. 5 for the horizontal database.

However, unless the RDF data set has a fixed structure and a small number of predicates, the row-based horizontal database is not practical for the following reasons. First, the table is very sparse; for instance, in Fig. 4, 38 out of 56 property cells are null. Second, a horizontal database is not scalable to the number of predicates. For example, the Wikipedia data set needs a table with over 50,000 columns, which is impractical for implementation. Third, this scheme does not handle data of dynamic structures well. When an RDF statement with a new predicate is added, the table has to be expanded, which requires costly data restructuring

subject	*degree*	*enrolls*	*title*	*supervisor*	*interest*	*teacher*	*code*
⟨Tom⟩	PhD		Instructor	⟨Sam⟩			
⟨May⟩	MPhil	⟨Db⟩		⟨Sam⟩			
⟨Roy⟩	BSc	⟨Db⟩, ⟨Web⟩					
⟨Sam⟩			Professor		Database, Datamining		
⟨Kat⟩			Professor		Security		
⟨Db⟩						⟨Sam⟩	C123
⟨Net⟩						⟨Sam⟩	C246
⟨Web⟩						⟨Tom⟩	C135

Figure 4: Horizontal database for Fig. 1

```
SELECT subject FROM horizontal_table
WHERE degree = "MPhil" AND enrolls = "<Db>"
  AND supervisor = "<Sam>";
```

Figure 5: SQL statement for triple store

operations.

2.3. Vertical Database

Abadi et al.[2] proposed to store all objects of each different predicate in a separate vertical table. In this vertical partitioning approach, Fig. 1 requires 7 tables as shown in Fig. 6.[1] The advantages of this scheme are as follows. (1) It is simple to implement. (2) It does not waste space on null values (but it does waste space replicating the subject columns in all tables). (3) It does not store multiple values in one cell while not every RDBMS handles multi-valued fields. (4) Some column-based databases (e.g., C-Store[12] and MonetDB[13]), which are optimized for column-based storage, can be used.

degree table

subject	*property*
⟨Tom⟩	PhD
⟨May⟩	MPhil
⟨Roy⟩	BSc

code table

subject	*property*
⟨Db⟩	C123
⟨Net⟩	C246
⟨Web⟩	C135

enrolls table

subject	*property*
⟨May⟩	⟨Db⟩
⟨Roy⟩	⟨Db⟩
⟨Roy⟩	⟨Web⟩

supervisor table

subject	*property*
⟨Tom⟩	⟨Sam⟩
⟨May⟩	⟨Sam⟩

teacher table

subject	*property*
⟨Db⟩	⟨Sam⟩
⟨Net⟩	⟨Sam⟩
⟨Web⟩	⟨Tom⟩

title table

subject	*property*
⟨Tom⟩	Instructor
⟨Sam⟩	Professor
⟨Kat⟩	Professor

interest table

subject	*property*
⟨Sam⟩	Database
⟨Sam⟩	Datamining
⟨Kat⟩	Security

Figure 6: Vertical database for Fig. 1

However, this scheme also has some disadvantages. (1) To process queries that involve multiple predicates, table-joins are always needed. For example, the SPARQL query in Fig. 2 is translated into the SQL query in Fig. 7 on the vertical database, which requires 2 joins. This scheme neglects any data and query patterns, which may suggest organizing some related predicates in one table can save table-joins. For example, resources ⟨Db⟩, ⟨Net⟩, and ⟨Web⟩ about courses all have predicates ⟨teacher⟩ and ⟨code⟩, so they can be stored together in the same table. (2) Column-based databases are not as mature and popular as row-based RDBMS. (3) Sidirourgos et al.[3] pointed out the scalability issue of this scheme. They found that the triple store approach outperformed the vertical partitioning approach on a row-based RDBMS in answering queries. Although the vertical database performed much faster on a column store than on an RDBMS, the query performance decreased rapidly as the number of predicates increased. The triple-store could still outperform the vertical database on a column store when the number of predicates was large enough, e.g., around 200 predicates.

[1]Abadi et al. used IDs to represent subjects and predicates.

```
SELECT a.subject FROM degree a, enrolls b, supervisor c
WHERE a.subject = b.subject AND b.subject = c.subject
AND a.value = "MPhil" AND b.value = "<Db>"
AND c.value = "<Sam>";
```

Figure 7: SQL statement for vertical database

2.4. Property Tables

The property table approach proposed by the Jena project[4] strikes a balance between the horizontal approach and the vertical approach. Correlated predicates are grouped into a property table. Each subject with any of the predicates of this property table has one row in the table. As shown in Fig. 8, 7 predicates are clustered into 3 property tables: *student*, *staff*, and *course*.[2] In the *course* table, all the resources ⟨Db⟩, ⟨Net⟩, ⟨Web⟩ have both ⟨teacher⟩ and ⟨code⟩ predicates. However, in the *student* table, 2 out of 3 subjects do not have all the predicates maintained by the table. Also note that the subject ⟨Tom⟩ appears in both *student* and *staff* tables. Obviously, the proportion of null cells (only 3 nulls out of 21 predicate cells) is largely reduced when compared to the horizontal database. In contrast to the vertical database, no table join is needed when a query only involves the predicates within the same table. For example, the SPARQL query in Fig. 2 is translated into the SQL statement in Fig. 9, which needs no table-join. However, if a query involves the predicates from different property tables, table-joins cannot be avoided. In general, some table joins can often be saved compared

[2]Jena actually uses a resource table and a literal table to reference the subjects and objects by system IDs in property tables.

staff table

subject	*title*	*interest*
⟨Tom⟩	Instructor	
⟨Sam⟩	Professor	Database, Datamining
⟨Kat⟩	Professor	Security

student table

subject	*degree*	*enrolls*	*supervisor*
⟨Tom⟩	PhD		⟨Sam⟩
⟨May⟩	MPhil	⟨Db⟩	⟨Sam⟩
⟨Roy⟩	BSc	⟨Db⟩, ⟨Web⟩	

course table

subject	*teacher*	*code*
⟨Db⟩	⟨Sam⟩	C123
⟨Net⟩	⟨Sam⟩	C246
⟨Web⟩	⟨Tom⟩	C135

Figure 8: Property tables for motivating example

```
SELECT subject FROM student
WHERE degree = "MPhil" AND enrolls = "Db"
  AND supervisor = "Sam";
```

Figure 9: SQL statement for property tables

with the vertical database. Oracle also provides a property-table-like facility called *subject-property matrix* table.[14]

2.5. Problems of Property Table Approach

Despite the above advantages of the property table scheme, it poses two technical challenges as described below.

Multi-valued properties: Some single predicate representing a one-to-many (or many-to-many) relationship between a subject and multiple objects (e.g., ⟨interest⟩ and ⟨enrolls⟩) can result in multiple objects stored in a single cell. Abadi et al. argued this is a drawback of the property table approach compared to the vertical approach because multi-valued fields were poorly supported by RDBMS.[2] Jena avoided this problem by not clustering multi-valued predicates in any property table but by storing them in a fallback triple store instead.[15] Another solution is to use the proprietary support of multi-valued fields by some RDBMS systems, e.g., PostgreSQL and Oracle.

Lack of good predicate clustering algorithms: The benefit of fewer joins can only be capitalized when the predicates are well clustered so that the transactions for a subject are contained within only one or few property tables. This is the major drawback pointed out by Abadi.[2] On the one hand, the schemas of the property tables can be designed manually. However, this requires the database designer to have a priori domain knowledge on the data and query patterns, which is often not feasible. For instance, Wikipedia adopts a liberal editing, collaborative authoring model where there are no governing *hard* schemas, but only *soft* guidelines[16]; therefore, the data pattern is hardly comprehensible by humans. On the other hand, the predicates can also be "somehow" clustered by a suitable data mining algorithm based on the data or query patterns to generate the property table schemas.

2.5.1. Data Pattern Analysis for Schema Design

Our ACTL approach aims to analyze the data patterns in order to automate schema design. Two related approaches can be applied for the same purpose. They are the classical frequent pattern mining approach suggested by Ding and Wilkinson[15, 17] and the HoVer approach proposed by Cui et al.[18] We compare these two approaches with ACTL in Sect. 8.

2.5.2. Query Pattern Analysis for Schema Design

Another approach to aid schema designs is by analysing the query patterns. Ding and Wilkinson proposed to mine frequent predicate patterns from query logs to form candidate property tables.[15] However, this technique suffers the same problems discussed in Sect. 8 about the usefulness of the frequent pattern mining results. Many research works[19–22] proposed different techniques to evaluate the query costs of database designs against a given query workload for database optimizations, e.g., index selection, table partitioning. However, in many RDF applications, query patterns are unknown and dynamic, which makes these techniques not applicable.

3. Problem Definition

This section models the property table design problem as the following attribute clustering problem. Given a set of attributes (predicates) and a set of transactions (subjects) associated with the attribute set, we intend to partition the attributes into *clusters*, each comprising the columns of one table, so that the proportion of null cells is not less than a given threshold while the number of tables (clusters) is minimized. Definition 1 formally defines the terminology we used in this paper.

Definition 1. A *database* $D = (A, T)$ consists of a set of *attributes* A and a set of *transactions* T. Each transaction $t \in T$ is associated with a group of attributes $\alpha(t) \subseteq A$. A database *schema* Π for D is a partition of A, i.e., (1) $\bigcup_{C \in \Pi} = A$, (2) $C \cap C' = \emptyset$ for any distinct $C, C' \in \Pi$, and (3) $C \neq \emptyset$ for any $C \in \Pi$. Each element C in Π is called an *attribute cluster*, for which the following properties are defined:

The *transaction group* of C (a subset of T):

$$\tau(C) = \{t \in T : \alpha(t) \cap C \neq \emptyset\} \tag{1}$$

The *table load* of C (an integer):

$$\mathrm{LD}(C) = \sum_{t \in \tau(C)} |\alpha(t) \cap C| \tag{2}$$

The *table load factor* of C (a rational number):

$$\mathrm{LF}(C) = \frac{\mathrm{LD}(C)}{|C| \times |\tau(C)|} \tag{3}$$

In our example, 8 subjects (i.e., ⟨Tom⟩, ⟨May⟩, ⟨Sam⟩, ⟨Kat⟩, ⟨Db⟩, ⟨Net⟩, and ⟨Web⟩) form the set of transactions while 7 predicates (i.e., ⟨degree⟩, ⟨enrolls⟩, ⟨title⟩, ⟨supervisor⟩, ⟨interest⟩, ⟨teacher⟩, ⟨code⟩) form the set of attributes. Fig. 4 and Fig. 8 organize these attributes in two different database schemas, where the attributes are organized in a single table (cluster), and in 3 tables respectively.

In Fig. 8, the *student* cluster contains 3 attributes: ⟨degree⟩, ⟨enrolls⟩, and ⟨supervisor⟩. The transaction group of *student* is { ⟨Tom⟩, ⟨May⟩, ⟨Roy⟩ }. The table load of an attribute cluster is the number of non-null attribute cells in a table, so the table load of *student* is 7. The table load factor of an attribute cluster is its table load divided by the total number of attribute cells, so the table load factor of *student* is $7/9 = 0.778$.

Definition 2 formally defines the *Attribute Clustering by Table Load (ACTL)* problem. Essentially, it is how to partition a given set of attributes into the fewest clusters such that each cluster has a table load factor not less than a given threshold θ. We have proved that the ACTL is NP-complete (Theorem 1), as shown in Appendix A. Fig. 8 shows an optimal solution to the example for $\theta = 2/3$.

Definition 2. Given (1) a database (A, T), and (2) a load factor threshold θ, where $0 \leq \theta \leq 1$, find a partition (schema) Π of A such that:

1. for each cluster $C \in \Pi$, $\mathrm{LF}(C) \geq \theta$, and
2. for any possible partition Π' of A where $\mathrm{LF}(C') \geq \theta$ for any $C' \in \Pi'$, $|\Pi| \leq |\Pi'|$.

Theorem 1. *The ACTL problem is NP-complete.*

4. Agglomerative Clustering

This section introduces a basic agglomerative algorithm to approximate the optimal solution to the ACTL problem, and shows that this algorithm has high time and space complexity. Algorithm 1 shows the agglomerative ACTL algorithm. At first, each attribute forms a distinct attribute cluster. A priority queue is used to maintain all *unordered* pairs of clusters $\{C, C'\} \in \Pi$ where (1) the *combined load factor* $\mathrm{LF}(C \cup C')$ (Definition 3) is not below the given load factor threshold θ, and (2) the cluster pair with the highest combined load factor is placed at the top of the queue. Then, the cluster pair from the queue with highest combined load factor is taken for merging into a new cluster. All existing cluster pairs with any of the old clusters that have already been merged are removed from the queue. The new cluster is then used to generate new cluster pairs with the existing clusters. Among these new cluster pairs, those with the combined load factor greater than or equal to θ are added into the queue. Iteratively, the next cluster pair is taken from the top of the queue to repeat the above steps until the queue is empty.

Definition 3. The *combined load factor* of two clusters C and C', where $C \cap C' = \emptyset$, is the load factor for the cluster $C'' = C \cup C'$:

$$\mathrm{LF}(C'') = \frac{\mathrm{LD}(C) + \mathrm{LD}(C')}{|\tau(C) \cup \tau(C')| \times (|C| + |C'|)}$$

Algorithm 1. BasicACTL

```
Input: database $(A, T)$, where $|A| = n, |T| = m$
Input: load factor threshold $0 \leq \theta \leq 1$
Output: partition $\Pi$ of $A$ such that $\mathrm{LF}(C) \geq \theta$ for
    any $C \in \Pi$
1:  initialize $\Pi = \{\{a_1\}, \{a_2\}, \ldots, \{a_n\}\}$
2:  create an empty priority queue $Q$, where each
    element is an unordered pair of clusters $\{C, C'\}$
    and the pair with the largest $\mathrm{LF}(C \cup C')$ is
    placed at the top
3:  for all $\{C, C'\}$ where $C, C' \in P$ do
4:    compute $\mathrm{LF}(C \cup C')$ /* $O(m)$ */
5:    if $\mathrm{LF}(C \cup C') \geq \theta$ then
6:      add $\{C, C'\}$ to $Q$ /* $O(\log |Q|)$ */
7:    end if
8:  end for/* $O(n^2(m + \log n))$ */
9:  while $Q$ is not empty do /* let $|\Pi| = p$ (i.e.,
    $O(n)$) and $|Q| = q$ (i.e., $O(n^2)$) */
10:   pop $\{C, C'\}$ from $Q$ /* $O(1)$ */
```

```
11:     $C'' \leftarrow C \cup C'$ /* O(n) */
12:     delete $C, C'$ from $\Pi$
13:     delete all pairs with $C$ or $C'$ from $Q$ /* $O(p \log q)$ */
14:     for all $\hat{C} \in \Pi$ do
15:         if LF$(C'' \cup \hat{C}) \geq \theta$ then /* $O(m)$ */
16:             add $\{C'', \hat{C}\}$ to $Q$ /* $O(\log q)$ */
17:         end if
18:     end for/* $O(p(m + \log q))$ */
19:     add $C''$ to $P$
20: end while/* $O(n^2 m + n^2 \log n)$ */
```

Here, we run Algorithm 1 using Fig. 1 with $\theta = 2/3$. Initially, there are 7 clusters, each containing one attribute and has a load factor 1. The clusters $\{\langle\text{teacher}\rangle\}$ and $\{\langle\text{code}\rangle\}$ are first merged into the new cluster $\{ \langle\text{teacher}\rangle, \langle\text{code}\rangle \}$ because their combined load factor is 1. Then, either $\{ \langle\text{title}\rangle, \langle\text{interest}\rangle \}$, $\{ \langle\text{degree}\rangle, \langle\text{enrolls}\rangle \}$, or $\{ \langle\text{degree}\rangle, \langle\text{supervisor}\rangle \}$ is created because they all have a combined load factor 5/6. Suppose $\{ \langle\text{title}\rangle, \langle\text{interest}\rangle \}$ and $\{ \langle\text{degree}\rangle, \langle\text{enrolls}\rangle \}$ are created. Finally, $\{ \langle\text{degree}\rangle, \langle\text{enrolls}\rangle, \langle\text{supervisor}\rangle \}$ is created by merging $\{ \langle\text{degree}\rangle, \langle\text{enrolls}\rangle \}$ and $\{ \langle\text{supervisor}\rangle \}$ while the combined load factor 7/9 is still not below $\theta = 2/3$. After that, no bigger cluster can be formed as merging any two clusters will make the combined load factor drop below θ.

4.1. Core Data Structure

To facilitate the computation of the table load factor, ACTL is conducted using an attribute-oriented data structure rather than a transaction-oriented one. The transactions, attributes, and clusters are assigned unique system IDs for all manipulations in clustering. For each cluster C, (1) a *sorted* list of the transaction IDs for the transaction group, and (2) the integer table load are maintained. Initially, since each cluster contains one distinct attribute, the IDs of all transactions containing that attribute are sorted into the transaction list for that cluster, and the table load for each cluster is the size of the list. (Note that the table load factors for all initial clusters are 1.)

When two clusters C and C' are combined into a new cluster $C'' = C \cup C'$, the transaction group of C'' is the union of the transaction groups of C and C', i.e., $\tau(C \cup C') = \tau(C) \cup \tau(C')$. Since the transaction lists for C and C' are sorted, they can be merged into the sorted transaction list for C'' in $O(k)$ time, where $k = \max\{|\tau(C)|, |\tau(C')|\}$. The table load for C'' is the sum of the table loads for C and C', i.e., $\text{LD}(C \cup C') = \text{LD}(C) + \text{LD}(C')$. The table load factor for C'' can then be easily computed by Eq. 3. Therefore, the time complexity to compute the combined load factor for two clusters from their transaction groups and table loads is $O(m)$, where m is the number of transactions.

4.2. Complexity Analysis

The time complexity of the non-trivial steps in Algorithm 1 are specified in the comments. The total time complexity of the algorithm is $O(n^2 m + n^2 \log n)$, where n is the number of attributes and m is the number of transactions. Clearly, the time complexity is more sensitive to the number of attributes than to the number of transactions. The space complexity of this algorithm is determined by (1) the storage for the transaction lists and attributes for each cluster, which is nm and (2) the priority queue size, which is $n(n-1)/2$ in the worst case. The total space complexity is $O(nm + n^2)$.

5. Pruning Techniques

We can see that Algorithm 1 may not be time and space efficient enough to handle large sets of attributes and transactions. The Wikipedia dataset we used in our experiments has around 50,000 attributes and 480,000 transactions, and was too large for the above algorithm to cluster in an acceptable time period. Our implementation of this basic algorithm failed with an insufficient memory error after processing the dataset for 3 hours as the priority queue had grown to exhaust the 12GB memory we used. In this section, we discuss several pruning techniques that can significantly improve the time and space efficiency of the algorithm. Our implementation of the algorithm enhanced with these pruning techniques succeeded in clustering the dataset in ∼42 minutes.

5.1. Transaction Group Equality Test

In real-life datasets, there are many clusters of attributes which share identical transaction groups. In Fig. 1, the attributes $\langle\text{teacher}\rangle$ and $\langle\text{code}\rangle$ have the same transaction group: $\{ \langle\text{Db}\rangle, \langle\text{Net}\rangle, \langle\text{Web}\rangle \}$. These attributes with the same transaction group will surely be merged into the same cluster because their combined load factor is always 1. We can use an efficient *transaction group equality test* to merge every set of attributes with the same transaction group into one cluster in the very beginning in order to reduce the total number of initial clusters.

Since testing the equality of two sets has non-trivial computation cost, we can use a simple hash function on transaction groups to efficiently find out all attributes *potentially* having the same transaction group. The hash value of an attribute is computed as the sum of the IDs of all transactions in its transaction group modulo the maximum hash value: $\text{HASH}(a) = \sum_{t \in \tau(a)} \text{ID}(t) \mod M$.

To further process each set of attributes with the same hash value, we divide them into subsets where all attributes in the same subset has the same number of transactions. The hash value and size comparisons on transaction groups can effectively filter out most of false positives. Nevertheless, we still need to exercise the actual equality test on the transaction groups for all attributes in each subset. Since the transaction groups are represented by sorted ID lists, the time to confirm if two lists are equal is linear to the list size. Algorithm 2 performs the above process.

Algorithm 2. ClusterAttrsWithEqualTrxGrps

Input: set of attributes A

Output: list L of attribute sets where all attributes in each set share the same transaction group

```
1: initialize $L$ to be an empty set
2: create a hash table $H$ of attributes sets, where $H(h) = \{a : \text{HASH}(a) = h\}$
3: for all attribute $a \in A$ do /* compute $H$ */
4:   Add $a$ to the attribute set $H(\text{HASH}(a))$
5: end for
6: for all attribute set $A'$ in $H$ do
7:   create a list $\bar{L}$ of attribute sets
8:   for all attribute $a' \in A'$ do
9:     $done \leftarrow false$
10:    for all attribute set $\bar{A} \in \bar{L}$ do
11:      let $a$ be some attribute in $\bar{A}$
12:      if $|\tau(\bar{a})| = |\tau(a')|$ then /* try to avoid comparing $\tau(\bar{a}) = \tau(a')$ */
13:        if $\tau(\bar{a}) = \tau(a')$ then
14:          put $\bar{a}$ into $\bar{A}$
15:          $done \leftarrow true$
16:          break the for-loop
17:        end if
18:      end if
19:    end for
20:    if $done = false$ then
21:      create a new set $A''$ in $\bar{L}$ and add $a'$ to $A''$
22:    end if
23:  end for
24:  append $\bar{L}$ to $L$
25: end for
```

5.2. Maximum Combined Load Factor

In Lines 14-19 of Algorithm 1, when a new cluster C'' is created, we need to find all existing clusters $C \in \Pi$ such that $\text{LF}(C'' \cup C) \geq \theta$ and put $\{C, C''\}$ into the priority queue. It is very time-consuming if we compute $\text{LF}(C'' \cup C)$ for all clusters $C \in \Pi$ as each list-merge on $\tau(C'') \cup \tau(C)$ costs $O(m)$ time. We have implemented a mechanism to prune all clusters $C \in \Pi$ where $\text{LF}(C'' \cup C)$ *cannot* reach θ without computing $\tau(C'') \cup \tau(C)$.

5.2.1. Bounds on Transaction Group Size To Attain Maximum Combined Load Factor

When we examine a new cluster C'' just created, we know its member attributes and transaction group size. For the combined load factor of C'' and some existing C (i.e., $\text{LF}(C'' \cup C)$), we can derive the upper bound before $\text{LF}(C)$ is known and $\tau(C'') \cup \tau(C)$ is computed. Note that $\text{LF}(C'' \cup C)$ attains its maximum value when (1) C is fully loaded and (2) $\tau(C'') \cup \tau(C)$ is the smallest, which equals $\max\{|\tau(C'')|, |\tau(C)|\}$.

$$\text{LF}(C'' \cup C) = \frac{\text{LD}(C'') + \text{LD}(C)}{|\tau(C'') \cup \tau(C)|\,(|C''| + |C|)} \tag{4}$$

$$\leq \frac{\text{LD}(C'') + |\tau(C)| \times |C|}{\max\{|\tau(C'')|, |\tau(C)|\}(|C''| + |C|)} \tag{5}$$

Therefore, if we require $\text{LF}(C'' \cup C) \geq \theta$, we must have:

$$\frac{\text{LD}(C'') + |\tau(C)| \times |C|}{\max\{|\tau(C'')|, |\tau(C)|\}(|C''| + |C|)} \geq \theta \tag{6}$$

Case 1: if $0 < |\tau(C)| \leq |\tau(C'')|$ then

$$\frac{\text{LD}(C'') + |\tau(C)| \times |C|}{|\tau(C'')|\,(|C''| + |C|)} \geq \theta \tag{7}$$

$$|\tau(C)| \geq \frac{|\tau(C'')|\,(|C''| + |C|)\theta - \text{LD}(C'')}{|C|} \tag{8}$$

Since $|\tau(C)|$ is a positive integer, we have:

$$|\tau(C)| \geq \left\lceil \frac{|\tau(C'')|\,(|C''|+|C|)\theta - \mathrm{LD}(C'')}{|C|} \right\rceil \quad (9)$$

Case 2: if $|\tau(C)| > |\tau(C'')|$ then

$$\theta \leq \frac{\mathrm{LD}(C'') + |\tau(C)| \times |C|}{|\tau(C)|\,(|C''|+|C|)} \quad (10)$$

$$|\tau(C)| \times ((|C''|+|C|)\theta - |C|) \leq \mathrm{LD}(C'') \quad (11)$$

Case 2a: if $|\tau(C)| > |\tau(C'')|$ and $\theta > \frac{|C|}{|C''|+|C|}$ then

$$|\tau(C)| \leq \frac{\mathrm{LD}(C'')}{(|C''|+|C|)\theta - |C|} \quad (12)$$

Case 2b: if $|\tau(C)| > |\tau(C'')|$ and $\theta \leq \frac{|C|}{|C''|+|C|}$ then Eq. 11 always hold since $\mathrm{LD}(C'')$ is positive. Therefore, we only need to consider Case 2a. Since $|\tau(C)|$ is a positive integer, we have:

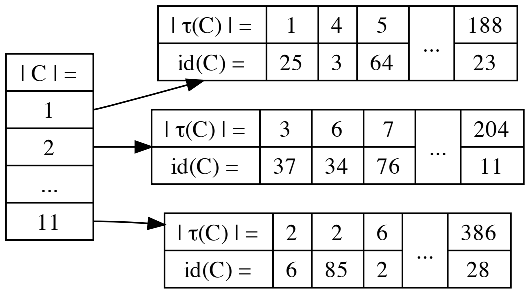
$$|\tau(C)| \leq \left\lfloor \frac{\mathrm{LD}(C'')}{(|C''|+|C|)\theta - |C|} \right\rfloor \quad (13)$$

For example, suppose the newly created cluster C'' has 4 attributes, a load factor of 0.9, and a transaction group of size 1,000 while $\theta = 0.8$. Among all existing clusters with 5 attributes, we only need to consider those with transaction group size from $\left\lceil \frac{1000\times(4+5)\times 0.8 - 1000\times 4\times 0.9}{5} \right\rceil = 720$ to $\left\lfloor \frac{1000\times 4\times 0.9}{(4+5)\times 0.8-5} \right\rfloor = 1636$ for merging with C''.

5.2.2. Cluster Sorter Data Structure

We have designed a data structure called *cluster sorter* to sort each cluster C using (1) its number of attributes $|C|$ as the primary key, and (2) the size of its transaction group $|\tau(C)|$ as the secondary key. This two-level sorting is supported by the data structure shown in Fig. 5.2.2. The aim of this data structure is to quickly find out, given integers n, m_{lower}, m_{upper}, all clusters C with n attributes and a transaction group of size between m_{lower} and m_{upper}.

The first-level sorting is designed to return a list of all clusters with a specified number of attributes in $O(1)$ time. This is implemented as an array of pointers where each pointer points to a list of cluster data nodes. The n-th array entry points to the list



of all clusters with n attributes. For example, the second list in Fig. 5.2.2 records all clusters with 2 attributes.

The second-level sorting is designed to find out all clusters of which the transaction group sizes are within a specified range. It is supported by a list of data nodes for all clusters with a particular number of attributes. Each node stores the transaction group size and the ID of a different cluster. On the list, the nodes are sorted by the transaction group sizes of the corresponding clusters. To efficiently return a sub-list of clusters with a specified range of transaction group size, each list is implemented as a red-black tree, which can locate the sub-list in $O(\log n)$ time. For example, in the second list of the example, two clusters with IDs 37 and 34 are returned if the requested range is between 2 to 6 inclusively.

5.2.3. Algorithm to Find Candidate Clusters for Merging

Based on Sect. 5.2.1 and Sect. 5.2.2, we have developed Algorithm 3 to efficiently find out all existing clusters C to merge with the newly created cluster C'' with a combined load factor $\mathrm{LF}(C'' \cup C) \geq \theta$.

Algorithm 3. FindCandidateClusters

```
Input: cluster sorter $S$ where $S(n, m_{lower}, m_{upper})$
    returns a list of existing clusters $C \in \Pi$ with
    $n$ attributes and transaction groups of size $m$,
    where $m_{lower} \leq m \leq m_{upper}$
Input: newly created cluster $C''$
Input: load factor threshold $\theta$
Output: list $L$ of existing clusters $C$ where
    $\mathrm{LF}(C'', C) \geq \theta$
1: initialize $L$ to be an empty list
2: for all $n \leftarrow 1\ldots$ maximum number of attributes of existing clusters in $P$ do
3:     $m_{lower} \leftarrow \lceil \min\{\frac{|\tau(C'')|(|C''|+n)\theta - \mathrm{LD}(C'')}{n}, \tau(C'')\} \rceil$
4:     if $\frac{n}{|C''|+|C|} < \theta$ then
```

```
5:      $m_{upper} \leftarrow \lfloor \max\{\frac{\mathrm{LD}(C'')}{(|C''|+n)x-n}, \tau(C'')+1\}\rfloor$
6:    else
7:      $m_{upper} \leftarrow \infty$
8:    end if
9:    $L \leftarrow S(n, m_{lower}, m_{upper})$ /* $O(\log n)$ */
10: end for
11: for all $C \in L$ do
12:   if $\frac{\mathrm{LD}(C'')+\mathrm{LD}(C)}{\max\{|\tau(C'')|,|\tau(C)|\}(|C''|+|C|)} \geq \theta$ then
13:     remove $C$ from $L$
14:   else
15:     compute $\tau(C'') \cup \tau(C)$ /* $O(\log m)$ */
16:     if $\frac{\mathrm{LD}(C'')+\mathrm{LD}(C)}{|\tau(C'')\cup\tau(C)|(|C''|+|C|)} < \theta$ then
17:       remove $C$ from $L$
18:     end if
19:   end if
20: end for
```

5.3. Iterative Clustering

When we cluster a large number of attributes using Algorithm 1, we face the insufficient memory problem when too many cluster pairs are stored in the priority queue. For example, there are over 50,000 attributes in the Wikipedia dataset, which can produce 1.25 billion possible attribute pairs at maximum. Initially, if all possible pairs of clusters, each having one attribute, are candidates for merging, the priority queue would need tens of gigabytes of memory space. This could not only run out of the memory but also significantly slow down the insertions of a new cluster pair to and the deletions of an invalid cluster pair from the priority queue. Each of these operations costs $O(\log q)$ time, where q is the queue size.

It is easy to see that the higher the load factor threshold is, the more effective Algorithm 3 is in pruning unqualified cluster pairs for merging, the fewer candidate cluster pairs which attain the threshold need to be added to the priority queue. Since Algorithm 1 merges the clusters from the highest combined load factor down to the required load factor threshold, we can do the clustering step by step.

However, this iterative clustering approach does not always improve the performance. The reason is that the combined load factor of some candidate cluster pairs may need to be *recomputed* for those pairs which cannot be pruned by Algorithm 3. For example, if clusters C and C' have a combined load factor $\mathrm{LF}(C \cup C') = 0.75$, there is a chance that the same $\mathrm{LF}(C \cup C')$ is recomputed in all three iterations at $\theta = 0.9, 0.8, 0.7$. Nevertheless, this approach allows our clustering algorithm to be tunable for speeding up performance as well as to be scalable for adapting to different data sizes.

5.4. Modified Clustering Algorithm

Combining the techniques discussed in Sect. 5.1, 5.2, and 5.3, Algorithm 1 is modified to Algorithm 4.

Algorithm 4. ModifiedACTL

```
Input: database $(A, T)$
Input: load factor threshold $0 \leq \theta \leq 1$
Input: threshold step $0 \leq \delta \leq 1$
Output: partition $\Pi$ such that $\mathrm{LF}(C) \geq \theta$ for any
    $C \in \Pi$
1: $\Pi \leftarrow$ ClusterAttrsWithEqualTrxGrps($A$)
2: create a cluster sorter $S$ and add all cluster $C \in$
   $\Pi$ to $S$
3: create an empty priority queue $Q$, where each
   element is an unordered pair of clusters $\{C, C'\}$
   and the pair with the largest $\mathrm{LF}(C \cup C')$ placed
   in the top of $Q$
4: for all $\theta' = 1-\delta, 1-2\delta, \ldots, 1-k\delta, \theta$ where
   $k\delta < \theta$ for $k = 1, 2, \ldots$ do
5:   for all $C \in \Pi$ do
6:     $L \leftarrow$ FindCandidateClusters($S, C, \theta'$)
7:     add $\{C', C\}$ to $Q$ for all $C' \in L$ and $C' \neq C$
8:   end for
9:   while $Q$ is not empty do
10:    pop $\{C, C'\}$ from $Q$
11:    $C'' \leftarrow C \cup C'$
12:    delete $C, C'$ from both $\Pi$ and $S$
13:    delete all pairs containing $C$ or $C'$ from $Q$
14:    $L \leftarrow$ FindCandidateClusters($S, C'', \theta'$)
15:    add $\{C', C''\}$ to $Q$ for all $C' \in L$
16:    add $C''$ to both $\Pi$ and $S$
17:  end while
18: end for
```

6. Attribute Connectivity

Some applications may require only "connected" attributes to be allocated in the same cluster. Intuitively, when two disjoint sets of attributes are not connected, there is not any transaction that contains some attributes from one set and some from the other set. In that case, putting two disconnected sets of attributes together in the same cluster will not save any table-joins but only waste storage on null values.

For example, in Fig. 10, two clusters C_1 and C_2 may be produced from clustering 7 attributes over

	a_1	a_2	a_3	a_4
t_1	x	x		
t_2		x	x	
t_3			x	x

$C_1 = \{a_1, a_2, a_3\}$
LF$(C_1) = 1/2$

	a_5	a_6	a_7
t_4	x	x	
t_5	x	x	
t_6			x

$C_2 = \{a_5, a_6, a_7\}$
LF$(C_2) = 5/9$

Figure 10: Example on attribute connectivity

6 transactions given load factor threshold $\theta = 0.5$. We can say that attributes a_1 and a_2 are connected because transaction t_1 contains both attributes and it is natural to put them in the same cluster C_1. Although there is no transaction containing both a_1 and a_3, they are both connected to a_2 so a_1 and a_3 are *indirectly* connected. Similarly, a_4 is also connected to a_1, a_2, a_3 too. Therefore, it is natural to put all of them in single cluster C_1. Conversely, for cluster C_2, it seems not very useful to put a_7 with a_5 and a_6 because a_7 is "orthogonal" to others, even though the load factor of C_2 attains the threshold.

Definition 4 formally defines this attribute connectivity relation. In fact, $*$-connectivity is a transitive closure on 1-connectivity. We can consider the attributes being the nodes in a graph where there is an edge between two attributes if they are 1-connected. It is easy to observe that all nodes in each connected component of the graph forms an equivalence class of $*$-connectivity while $*$-connectivity is a equivalence relation (Theorem 2).

Definition 4. Two attributes a and a' are said to be 1-*connected* if $\tau(a) \cap \tau(a') \neq \emptyset$. Two attributes a and a' are said to be k-*connected* if there exists another attribute a'' such that a and a'' are $(k-1)$-connected and a'' and a' are 1-connected. Two attributes a and a' are said to be *connected* or $*$-*connected* if there exists some integer k such that a and a' are k-connected.

Theorem 2. *$*$-connectivity is an equivalence relation.*

We call each equivalence class on $*$-connectivity a *connectivity class*. If attribute connectivity is required, we can modify Algorithm 1 or 4 as follows. We can assign each connectivity class with a unique ID (CCID) and associate each attribute with its CCID. Then, we add a new condition into the algorithm to restrict that only two clusters with the same CCID can be merged. The CCID assignment algorithm is listed in Algorithm 5.

Algorithm 5. AssignCCID

```
Input: a database $D = (A, T)$
Output: a mapping CCID$[a]$ returns the connectivity class id for attribute $a$
 1: initialize CCID$[a] \leftarrow 0$ for all $a \in A$
 2: create a mapping $K : \mathbb{Z}^+ \mapsto 2^A$ and initialize $K[i] = \emptyset$ for all $1 \le i \le n$
 3: $j \leftarrow 1$
 4: for all $t \in T$ do
 5:   $A_t \leftarrow \alpha(t)$
 6:   if CCID$[a] = 0$ for all $a \in A_t$ then
 7:     CCID$[a] \leftarrow j$ for all $a \in A_t$
 8:     $K[j] \leftarrow A_t$
 9:     $j \leftarrow j + 1$
10:   else if there exists non-empty sets $A'_t, A''_t$ where $A'_t \cup A''_t = A_t$ such that all attributes $a' \in A'_t$ with CCID$[a'] = 0$ and all attributes $a'' \in A''_t$ share the same CCID$[a''] = d > 0$ then
11:     CCID$[a'] \leftarrow d$ for all $a' \in A'_t$
12:     $K[d] \leftarrow K[d] \cup A'_t$
13:   else if there exists $a, a' \in A_t$ such that both CCID$[a]$, CCID$[a'] > 0$ and CCID$[a] \neq$ CCID$[a']$ then /* attributes in $A_t$ have two or more different CCIDs */
14:     for all $a \in A_t$ do
15:       if CCID$[a] = 0$ then
16:         CCID$[a] \leftarrow j$
17:         $K[j] \leftarrow K[j] \cup \{a\}$
18:       else if CCID$[a] \neq j$ then
19:         $d \leftarrow$ CCID$[a]$
20:         CCID$[a''] \leftarrow j$ for all $a'' \in K[d]$
21:         $K[j] \leftarrow K[j] \cup K[d]$
22:         $K[d] \leftarrow \emptyset$
23:       end if
24:     end for
25:     $j \leftarrow j + 1$
26:   end if
27: end for
```

7. Measuring Schema Fitness

This section proposes several metrics to measure the fitness of an ACTL-mined schema. Definition 5 formally defines *storage configuration* as well as mappings ACLUSTER and TCLUSTERS. ACLUSTER returns the cluster to which attribute a belongs while TCLUSTERS returns the set of clusters covered by a given transaction.

Definition 5. Let (D, Π) be a storage configuration, where $D = (A, T)$ is a database and Π is a partition of A.

Define mapping ACLUSTER : $A \mapsto \Pi$ so that ACLUSTER$(a) = C$ if $a \in C$.

Define mapping TCLUSTERS : $T \mapsto 2^{\Pi}$ so that TCLUSTERS$(t) = \{\text{ACLUSTER}(a) : a \in \alpha(t)\}$.

A single transaction may spread across multiple tables (e.g., transaction ⟨Tom⟩ in Fig. 8). The *average number of clusters per transaction (ACPT)* (Definition 6) is the total number of rows in all tables (or the sum of the numbers of clusters storing each transaction) divided by the total number of transactions. An ACPT value is inclusively between 1 and the total number of clusters. For example, ACPT = 2.3 means each transaction spreads across 2.3 tables on average. Therefore, if the ACPT is small then the number of table-joins required for a query is potentially small. The more structured the data pattern is, the smaller the ACPT is. The ACPT for Fig. 8 is $9/8 = 1.125$.

Definition 6. The *average number of clusters per transaction (ACPT)* of a storage configuration is defined as follows:

$$\text{ACPT}(D, \Pi) = \frac{\sum_{t \in T} |\text{TCLUSTERS}(t)|}{|T|}, \text{ where } D = (A, T)$$

The *aggregate load factor (ALF)* (Definition 7) measures the aggregate storage efficiency of all tables. It is the total number of non-null cells in all tables divided by the total number of cells of the attribute columns in all tables. An ALF is inclusively between the load factor threshold and 1. The higher the ALF is, the less storage space is wasted on storing null values. The ALF for Fig. 8 is $18/21 = 0.857$.

Definition 7. The *aggregate load factor (ALF)* of a storage configuration is defined as follows:

$$\text{ALF}(D, \Pi) = \frac{\sum_{C \in \Pi} \text{LD}(C)}{\sum_{C \in \Pi}(|C| \times |\tau(C)|)}, \text{ where } D = (A, T)$$

Since good clustering is indicated by a low ACPT and a high ALF, we may use the ratio of the ALF/ACPT to find a suitable load factor threshold that produces a schema for balanced query performance and storage efficiency.

8. Other Clustering Approaches

As discussed in Sect. 2.5.1, some frequent pattern mining algorithm and the HoVer (horizontal representation over vertically partitioned subspaces) algorithm[18] can be used to cluster attributes for designing property tables. However, both algorithms have some shortcomings when they are used for this purpose.

8.1. Frequent Patterns Mining

Ding and Wilkinson[15, 17] suggested that frequent patterns mining techniques (e.g., Apriori[23] and FP-Growth[24]) could be used to *aid* property tables design. To verify this suggestion, we ran the FP-Growth algorithm[24] to find the frequent attribute patterns given different support values for the Barton Libraries dataset. The experimental result is given in Section 9.3.1, which shows that the algorithm cannot generate proper schema designs for the following reasons:

1. The frequent patterns are overlapping attribute subsets instead of disjoint clusters. Also, a frequent pattern only counts those transaction where all attributes in the pattern always occur together. Therefore, it is difficult to set a suitable support value while the frequent attribute subsets with the highest support may not be good table schemas. This problem can be illustrated in Fig. 8. The attribute subset {*degree*, *enrolls*, *supervisor*} in the *student* table does not receive the highest support (which is only 1 because of the subject *May*). However, we tend to group these attributes together because all three subjects in the table share at least two of the predicates.
2. There is no guarantee that the union of all frequent attribute subsets must cover the entire set of attributes. Even when a very low support value of 2 transactions was used to mine the frequent patterns from the Barton dataset, all the frequent attribute subsets cover only 149 out of 285 attributes.
3. For the above reasons, the frequent attribute subsets cannot be used to create property tables without manual selection and adjustments. We doubt how much the mining result can help design the schemas. It is infeasible for humans to analyze so many frequent patterns generated. When the support value was set as high as 100,000, 10,285 frequent patterns

(covering only 25 attributes) from the Barton dataset were still discovered.

In contrast, ACTL, by definition, produces disjoint attribute clusters covering the whole attribute set. This means that the produced clusters can be readily used as a schema design for property tables. The choice of the loading factor threshold only affects the number of property tables to be built and the storage efficiency of those tables when data are populated. In this sense, the ACTL approach is more suitable than the frequent pattern mining approach for mining property table schemas from data patterns.

8.2. HoVer Clustering

Cui et al. proposed a approach called *horizontal representation over vertically partitioned subspaces (HoVer)* to solve a similar problem addressed by ACTL.[18] HoVer is a basic hierarchical clustering algorithm on attributes using a new distance function, called *correlated degree*. The correlated degree between two attributes (called *dimensions*) a and a' is the size of the intersection of their transaction groups (called *active tuples*) divided by the size of the union of their transaction groups , i.e., $|\tau(\{a\}) \cap \tau(\{a'\})| \,/\, |\tau(\{a\}) \cup \tau(\{a'\})|$. A *correlation degree threshold* is given to the algorithm; HoVer seeks to group attributes into clusters (called *subspaces*) where for *every* pair of attributes in each cluster, the correlation degree is not less than the threshold. This algorithm poses the following problems when used to generate property tables schemas:

1. The correlation degree is merely a heuristic function without an obvious physical meaning like the load factor. It is difficult to use it to control the schema fitness.
2. The algorithm favors only small clusters but not large ones. In order for an unclassified attribute to be grouped into a existing cluster, HoVer requires the correlation degree threshold between that attribute with *every* attribute from that cluster to attain the correlation degree threshold. In other words, an unclassified attribute cannot be grouped into that cluster simply when merely a single attribute in that cluster is not sufficiently correlated with that unclassified one. Therefore, it is difficult to grow large clusters. Sect. 9.3.2 shows that, in our experiment where HoVer was used to cluster Wikipedia dataset, the average cluster size was only ∼3 and ∼15,000 clusters remained at very small thresholds. In contrast, ACTL progressively grew large clusters from merging small ones as the threshold was decreasing.

9. Experiments

We have conducted three experiments to show the applicability and effectiveness of our schema mining approach compared with other techniques. Experiment 1 performed ACTL (Algorithm 4) on two datasets, namely Wikipedia Infobox and Barton Libraries, for analyzing the clustering performance and the schema fitness. Experiment 2 performed clustering techniques (i.e., FP-Growth, and HoVer) on the Wikipedia dataset for comparison with ACTL. Experiment 3 stored the Wikipedia dataset using (1) property tables based on a schema mined from ACTL, (2) property tables based on a schema mined from HoVer, (3) a triplet store, and (4) a vertical database. These experiments were conducted on a Debian Linux (amd64) server with two Quad-Core Xeon Pro E5420 2.50GHz CPUs and 16GB RAM. Algorithm 4 was implemented as a single-threaded Java program, running on the Java HotSpot 64-Bit Server Virtual Machine initialized with 12GB heap-size.

9.1. Datasets

Wikipedia Infobox dataset. This dataset was extracted from the Wikipedia database[5] (20071018 English version). (The data extraction toolkit has been open-souced.[25]) Many Wikipedia pages contained an infobox created from a list of name-value pairs (e.g., `bridge_name` = `Tsing Ma Bridge`). Fig. 11 shows the infobox on the Wikipedia page "Tsing Ma Bridge." Every page with an infobox was treated as a subject resource, and each infobox field name qualified by the infobox name as a predicate. The infobox field value produced the objects for the predicate. If the field value did not have links, the object was a literal (e.g., "`HK$30 (cars)`" in the `toll` field). If the field value contained some links to other pages, each linked page was treated as one object resource, and the whole text value was treated as a literal. For example, in Fig. 11, the field `locale` generates the following 3 triplets:

```
<Tsing Ma Bridge> <Bridge#locale> <Ma Wan> .
<Tsing Ma Bridge> <Bridge#locale> <Tsing Yi Island> .
<Tsing Ma Bridge> <Bridge#locale>
  "Ma Wan Island and Tsing Yi Island" .
```

The conversion of the infobox data from the Wikipedia source to RDF required substantial data cleansing procedures, e.g., removal of comments, formatting tags, null values, and syntax errors from field names and values. This dataset contained 50,404 attributes (predicates), 479,599 transactions (subjects), and 9,549,024 RDF triplets.

Barton Libraries dataset. This dataset was provided by the Smile Project, which developed tools for library data management.[6] It was a collection of RDF statements converted from the MIT Libraries Barton catalog data. It was also used in the experiments by Sidirourgoes[3], Abadi[2] and Neumann[26]. This dataset contained 285 attributes, 3,811,592 transactions, and 35,182,174 triplets.

We selected these two datasets because they were huge and from real-life applications. While the Wikipedia dataset contained a large number of attributes, the Barton dataset contained a large number of transactions. Therefore, we could compare the experimental results of ACTL handling very large datasets of different natures.

9.2. ACTL Performance and Schema Fitness

In this experiment, we ran Algorithm 4 on both datasets with load factor thresholds from 1 to 0 at intervals of 0.2. Also, we clustered the Wikipedia dataset twice, one time with attribute connectivity disabled and another time with connectivity enabled. In the Barton dataset, all attributes are connected so ACTL gives the same result no matter attribute connectivity is enabled or not.

Fig. 12 and Fig.13 show the number of clusters and the execution time when ACTL was used to cluster the Wikipedia dataset and Barton Libraries dataset respectively at thresholds from 1 to 0. The experimental results are summarized below:

1. The Wikipedia dataset and the Barton dataset have 50,404 attributes and 285 attributes, requiring the same numbers of tables in vertical databases. When we clustered these datasets at load factor threshold 0.9, the numbers of clusters were 26,435 (for disabled attribute connectivity) and 26,449 (for enabled attribute connectivity) for the Wikipedia dataset, and 255 for the Barton dataset. This illustrates, at high thresholds, ACTL can already reduce the numbers of tables substantially.
2. Algorithm 4 (with pruning) took ∼42 minutes to cluster the Wikipedia dataset at threshold 0.0 with or without attribute connectivity enabled and only about one minute to cluster the Barton dataset at threshold 0.0. When we ran Algorithm 1 (ACTL without pruning) on the Wikipedia dataset, it raised the out of memory error after running for over 3 hours under the same setup. This demonstrates our pruning techniques are effective to make ACTL efficient enough for large datasets. Also, the results show that ACTL's complexity is more sensitive to the number of attributes than to the number of transactions.
3. The number of clusters for the Wikipedia dataset produced by ACTL with attribute connectivity enabled was 1,501. This is the number of attribute connectivity classes for this dataset. The Barton dataset has only one

```
{{Infobox Bridge
 |bridge_name=Tsing Ma Bridge
 |image=Tsing Ma Bridge 2008.jpg
 |caption=Tsing Ma Bridge at night
 |official_name=Tsing Ma Bridge
 |also_known_as=
 |carries=6 lanes of roadway (upper)<br>2 [[MTR]] rail tracks,
 2 lanes of roadway (lower)
 |crosses=[[Ma Wan Channel]]
 |locale=[[Ma Wan|Ma Wan Island]] and [[Tsing Yi Island]]
 |design=Double-decked [[suspension bridge]]
 |mainspan={{convert|1377|m|ft|0}}
 |width={{convert|41|m|ft|0}}
 |clearance={{convert|62|m|ft|0}}
 |open=[[April 27]], [[1997]]
 |toll=HK$30 (cars)
 |coordinates={{coord|22|21|05|N|114|04|27|E|
 region:HK_type:landmark}}
}}
```

Tsing Ma Bridge

Tsing Ma Bridge at night

Official name	Tsing Ma Bridge
Carries	6 lanes of roadway (upper) 2 MTR rail tracks, 2 lanes of roadway (lower)
Crosses	Ma Wan Channel
Locale	Ma Wan Island and Tsing Yi Island
Design	Double-decked suspension bridge
Width	41 metres (135 ft)
Longest span	1,377 metres (4,518 ft)
Vertical clearance	62 metres (203 ft)
Opening date	April 27, 1997
Toll	HK$30 (cars)
Coordinates	22°21′05″N 114°04′27″E

Figure 11: Wikipedia Infobox data

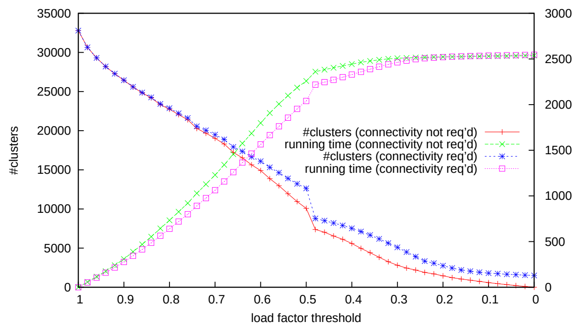


Figure 12: Performance of ACTL with pruning (Wikipedia Infobox dataset)

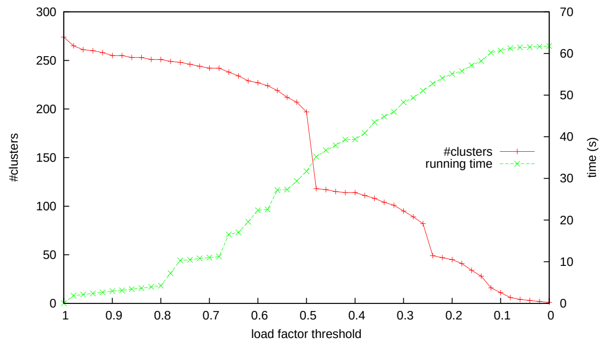


Figure 13: Performance of ACTL with pruning (Barton Libraries dataset)

attribute connectivity class, meaning all attributes are connected.

Fig. 14 and Fig. 15 show the ALFs, ACPTs, and ALF/ACPT ratios of using ACTL to cluster the Wikipedia dataset and Barton dataset respectively at different thresholds. The experimental results are summarized below:

1. Without ACTL, on average, a transaction (subject) has 14.26 and 5.647 attributes (not shown on the graphs) in the Wikipedia dataset and the Barton dataset respectively. In other words, a transaction on average spreads across ∼14 and ∼6 tables in vertical databases. The ACPTs dropped very quickly at high thresholds for both datasets. At the threshold 0.9, the ACPTs for the Wikipedia dataset and the Barton datasets were 5.60 and 2.43 respectively. This indicates that the ACTL-mined schemas at high thresholds can already contain transactions within small numbers of property tables.
2. The schemas mined with ACTL with attribute

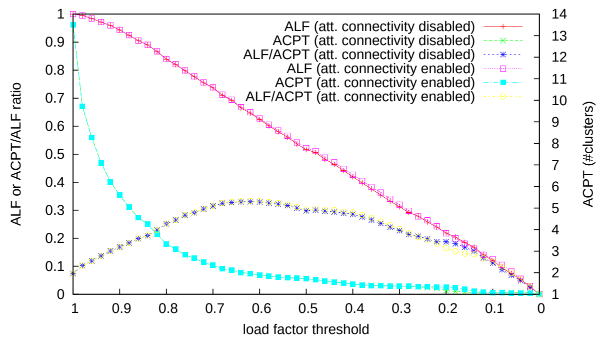


Figure 14: Schema fitness (Wikipedia infobox dataset)

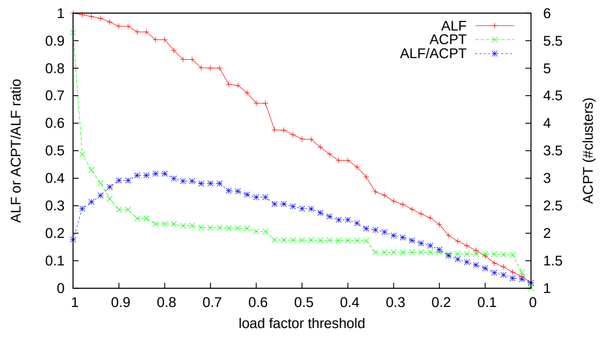


Figure 15: Schema fitness (Barton Libraries dataset)

connectivity disabled and enabled gave very similar ALF, ACPT, and ALF/ACPT values.

3. The ALF/ACPT ratios attained the highest for the Wikipedia dataset and the Barton dataset at thresholds 0.64 and 0.80, which generated property tables' schemas with balanced storage efficiency and query performance.
4. The ALFs were fairly proportional to and slightly higher than the load factor thresholds, so it is easy to use the load factor threshold to control the ALF.

9.3. Other Attribute Clustering Approaches

In this experiment, we attempted to cluster the Wikipedia dataset using FP-Growth and HoVer.

9.3.1. Frequent Patterns Mining

Table 1 summarizes the mining results of running the FP-Growth algorithm on the Barton dataset. The support values ranged from 2 to 500,000. The number of frequent attribute patterns decreased with increasing support values. However, there were two problems when these results were used to design the property tables. On the one hand, even

supp. value	#freq. patterns	#cov. atts	%cov. atts	supp. value	#freq. patterns	#cov. atts	%cov. atts
2	55,459,593	149	52.28	1,000	261,451	53	18.60
3	25,274,889	119	41.75	5,000	76,755	39	13.68
5	13,220,483	103	36.14	10,000	43,375	31	10.88
10	6,896,435	91	31.93	50,000	16,429	25	8.77
50	2,261,295	73	25.61	100,000	10,285	25	8.77
100	1,393,295	70	24.56	500,000	3,105	18	6.32
500	522,993	61	21.40				

Table 1: Frequent patterns generated by FP-Growth

when the support value was set to 100,000, there were still over 10,000 overlapping frequent patterns. On the other hand, only about half of the entire attribute set were covered even when the support value was set as low as 2. Therefore, it was difficult to find a right support value that could generate a good property tables schema. These problems are elaborated in Sect. 8.1.

9.3.2. HoVer Clustering

We implemented the HoVer algorithm in Java. We repeatedly performed HoVer on the Wikipedia dataset at correlation degree thresholds from 1.0 to 0.0 at intervals of 0.2, and collected numbers of clusters, average cluster sizes, ALFs, ACPTs, ALF/ACPT ratios for each clustering result. To particularly look into the trends of the above values for thresholds approaching zero, we performed HoVer 19 more times at thresholds from 0.019 to 0.001 at intervals of 0.001. Fig. 16 compares the number of clusters and the average cluster size at different thresholds between HoVer and ACTL. We can see that the rate of decrease in the number of clusters for HoVer was slower than that for ACTL. When the correlation degree threshold approached zero (e.g., 0.001), 15,375 clusters remained; when the threshold was exactly zero, there was only 1 cluster. In addition, the average cluster size for HoVer even remained small even at very small thresholds, e.g., only 3.303 at threshold 0.001. In contrast, ACTL progressively merged small clusters into large ones as the load factor threshold decreased. These phenomena are explained in Sect. 8.2.

9.4. Query Performance Comparison

In this experiment, we compared the actual query performance of four storage schemes: (1) a triple-store, (2) a vertical database, (3) HoVer-mined property tables and (4) ACTL-mined property tables. To make a fair comparison, we used the same mainstream database PostgreSQL, instead of different systems specialized for particular schemes.

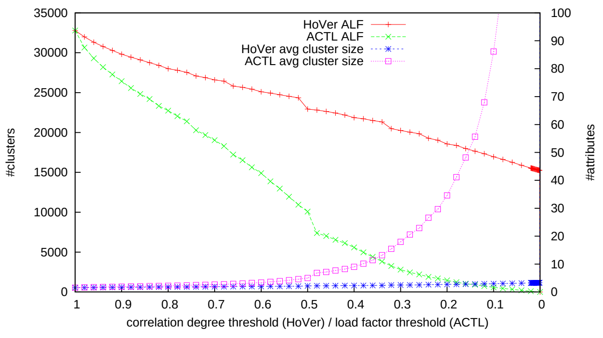


Figure 16: Number of clusters and ALF against threshold for HoVer

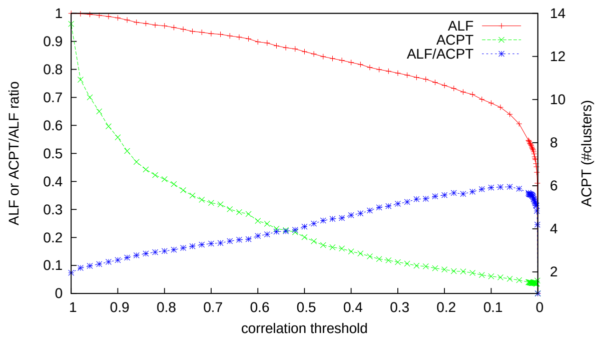


Figure 17: ALF, ACPT and ALF/ACPT ratio against threshold for HoVer

We used the Wikipedia dataset as it was commonly used for semantic (SPARQL) queries[27, 28]. Also, we designed 8 queries that were typical questions about the infobox data. Some queries were simple while others were more complex (which might require more table-joins). Yet, these queries are not trivial, which cannot be easily answered by typical search engines on Wikipedia pages. See Appendix B for details.

We loaded all 9,549,024 triplets into each database. The triple-store was a single table with three columns: subject ID, predicate ID, and object ID, with each triplet occupying one row. The vertical database was created with 50,404 predicate tables, each comprising one subject id column and one object ID column. The property-table databases were organized as follows. On the one hand, the schema mined by ACTL (with attribute connectivity disabled) at load factor threshold 0.7 was selected to make property tables. The ALF of the property tables at this threshold was 0.7367. On the other hand, the schema mined by HoVer at correlation degree threshold 0.19 was selected to

query	time (ms)				#joins			
	TS	Vert	HoVer	ACTL	TS	Vert	HoVer	ACTL
Q1	0.368	0.246	0.200	0.209	1	1	0	0
Q2	0.575	0.433	0.360	0.204	2	2	1	0
Q3	4.207	0.524	0.215	0.234	3	3	0	0
Q4	0.634	0.451	0.228	0.228	4	4	0	0
Q5	1.847	0.895	1.109	0.564	2	2	2	1
Q6	0.477	0.349	0.285	0.285	3	3	1	1
Q7	6.086	4.007	1.168	1.138	7	7	3	3
Q8	0.557	0.373	0.683	0.691	1	1	1	1

Table 2: Query performance of triple store (TS), vertical database (Vert) and HoVer-mined property tables (HoVer), ACTL-mined property tables (ACTL)

make property tables. This was because the ALF (storage efficiency) of the HoVer property tables at this threshold was 0.7365, which was slightly smaller than that of the ACTL property tables. In these property-table databases, each single-valued attribute in a cluster with more than one attribute was organized in some property table. Each property table comprised a subject ID column and the object ID columns for the predicates in the cluster. Each attribute which was multi-valued or in a single-attribute cluster was organized in a left-over triple-store. This measure was taken to handle the multi-valued field issue and reduce the number of tables; however, this could impact the query performance of these property tables. All columns containing IDs in all tables were indexed in each database.

To assess the query performance of each case, each query was translated into an independent SQL statement for each database design. The SQL statements and ID mappings are shown in Appendix B for details. Each query was executed 100 times, and the total elapsed time was recorded and averaged. The average query time per execution and the number of joins required by each query are tabulated in Table 2. The query time was generally proportional to the number of joins; in general, more time was required for a query with more joins. The query performance of the triple-store was the poorest, Although the vertical database required the same number of joins as the triple store did, the vertical database performed better because it involved joins between much smaller tables. The HoVer and ACTL databases performed better than the other databases. The ACTL database performed slightly better than the HoVer database as the ACTL database required fewer joins than the HoVer database for Q2 and Q5 and answered these queries faster.

10. Conclusions

In this paper, we have proposed a new data mining technique called Attribute Clustering by Table Load (ACTL) to handle RDF storage. This technique can be used to cluster predicates according to the data pattern to generate a property-table database schema that can balance storage efficiency and query performance. Experiments show that our algorithm can efficiently mine good property-table schemas from huge RDF datasets, while the derived property tables generally give better query performance over a triple store and a vertical database.

As future work, we are developing a hybrid approach by combining the triple store, vertical database, and property table schemes. While these three schemes have their own advantages in different situations, we are studying how to cluster a dataset into different partitions according to data and query patterns, and to manage each partition using the most suitable scheme. The attribute clusters can be used to create materialized views on top of a triple store or vertical database to reduce the chance of joining tables when answering queries. Another possible extension of ACTL is to infer RDF Schema by analyzing the patterns of an RDF dataset statistically.

References

[1] G. Klyne, J. J. Carroll, Resource Description Framework (RDF): Concepts and Abstract Syntax, Tech. rep., W3C (2004).

[2] D. J. Abadi, A. Marcus, S. R. Madden, K. Hollenbach, Scalable Semantic Web Data Management Using Vertical Partitioning, in: Proc. VLDB, 2007.

[3] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, S. Manegold, Column-Store Support for RDF Data Management: Not All Swans Are White, in: Proc. VLDB, 2008.

[4] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, K. Wilkinson, Jena: Implementing the Semantic Web Recommendations, in: Proc. WWW, 2004.

[5] Wikipedia:Database Download, http://en.wikipedia.org/wiki/Wikipedia_database.

[6] D. J. Abadi, A. Marcus, S. R. Madden, K. Hollenbach, Using the Barton Libraries Dataset as an RDF benchmark, Tech. rep., MIT (2007).

[7] T. Berners-Lee, Notation 3 (N3) A Readable RDF Syntax, Tech. rep., W3C (2008).

[8] J. Rusher, Triple Store, http://www.w3.org/2001/sw/Europe/events/20031113-storage/positions/rusher.html (2003).

[9] K. Wilkinson, C. Sayers, H. Kuno1, D. Reynolds, Efficient RDF Storage and Retrieval in Jena2, in: Proc. SWDB, Vol. 3, 2003.

[10] E. PrudâĂŹhommeaux, A. Seaborne, SPARQL Query Language for RDF, Tech. rep., W3C (2008).

[11] Z. Pan, J. Heflin, DLDB: Extending Relational Databases to Support Semantic Web Queries, Tech. rep., Department of Computer Science, Lehigh University (2004).
[12] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. OâĂŹNeil, P. OâĂŹNeil, A. Rasin, N. Tran, S. Zdonik, C-Store: a Column-Oriented DBMS, in: Proc. VLDB, 2005.
[13] MonetDB/SQL, http://monetdb.cwi.nl/SQL.
[14] E. I. Chong, S. Das, G. Eadon, J. Srinivasan, An Efficient SQL-based RDF Querying Scheme, in: Proc. VLDB, 2005.
[15] L. Ding, K. Wilkinson, C. Sayers, H. Kuno, Application-specific Schema Design for Storing Large RDF datasets, in: Proc. PSSS, 2003.
[16] S. Lee, P. Yee, T. Lee, D. W. Cheung, W. Yua, Descriptive Schema: Semantics-based Query Answering, in: SemWiki, 2008.
[17] K. Wilkinson, Jena Property Table Implementation, in: Proc. SSWS, 2006.
[18] B. Cui, J. Zhao, D. Yang, Exploring Correlated Subspaces for Efficient Query Processing in Sparse Databases, Proc. TKDE.
[19] S. Agrawal, V. Narasayya, B. Yang, Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design, in: Proc. SIGMOD, 2004.
[20] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, S. Fadden, DB2 Design Advisor: Integrated Automatic Physical Database Design, in: Proc. VLDB, 2004.
[21] S. Papadomanolakis, D. Dash, A. Ailamaki, Efficient Use of the Query Optimizer for Automated Database Design, in: Proc. VLDB, 2007.
[22] S. Agrawal, E. Chu, V. Narasayya, Automatic Physical Design Tuning: Workload as a Sequence, in: Proc. SIGMOD, 2006.
[23] R. Agrawal, R. Srikant, Fast Algorithms for Mining Association Rules in Large Databases, in: Proc. VLDB, 1994.
[24] J. Han, J. Pei, Y. Yin, R. Mao, Mining Frequent Patterns Without Candidate Generation: A Frequent-pattern Tree Approach, Data Mining and Knowledge Discovery.
[25] J. K. Chiu, T. Y. Lee, S. Lee, H. H. Zhu, D. W. Cheung, Extraction of RDF Dataset from Wikipedia Infobox Data, Tech. rep., Department of Computer Science, The University of Hong Kong (2010).
[26] T. Neumann, G. Weikum, RDF-3X: a RISC-Style Engine for RDF, in: Proc. VLDB, 2008, pp. 647–659.
[27] S. Auer, J. Lehmann, What have innsbruck and leipzig in common? extracting semantics from wiki content, in: ESWC, 2007.
[28] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, Z. G. Ives, Dbpedia: A nucleus for a web of open data, in: ISWC, 2007.
[29] I. Holyer, The NP-completeness of Some Edge-Partition Problems, SIAM J. Comput. 10 (4) (1981) 713–717.

Appendices

Appendix A. Proof of NP-Completeness of ACTL

To prove ACTL is NP-complete, we define the following *decisive version of ACTL* in Definition 8, which is same as Definition 2 in complexity class.

Definition 8. Given a database $D = (A, T)$, a load factor threshold $0 \leq \theta \leq 1$, and a positive integer k, Is there any partition Π of A such that $|\Pi| \leq k$ and $\mathrm{LF}(C) \geq \theta$ for each $C \in \Pi$? Define that $\mathrm{ACTL}(A, T, \theta, k)$ is true if the answer to this question is "yes."; otherwise, $\mathrm{ACTL}(A, T, \theta, k)$ is false.

Theorem 3. *The ACTL problem (Definition 8) is NP-complete.*

Proof. ACTL is in NP since a possible solution to ACTL is verifiable in polynomial time. We are going to show that ACTL is NP-hard by reducing an NP-complete problem called *edge-partition*[29]. The edge-partition problem $\mathrm{EP_n}$ is defined as follows. Given a graph $G(V, E)$, where V is its set of vertices, E is its set of edges, and $e = \{v_1, v_2\} \subseteq V$ for each $e \in E$. $\mathrm{EP_n}(G(V, E))$ is true if and only if E be partitioned into $E_1, \ldots, E_m$ such that each E_i for $1 \leq i \leq m$ generates a subgraph of G isomorphic to the complete graph K_n on n vertices. Holyer[29] proved that $\mathrm{EP_n}$ is NP-complete for any fixed integer $n \geq 3$. In the following, we show that $\mathrm{EP}_3(G(V, E))$ is polynomial time reducible to $\mathrm{ACTL}(A, T, \theta, k)$.

We can assume $|E| = 3m$, where $m \in \mathbb{Z}^+$. Since K_3 contains 3 edges, if $|E|$ is not a multiple of 3, $\mathrm{EP}_3(G(V, E))$ must be false. We construct the reduction as follows. (1) $A = E$. (2) $T = V$. (3) $\alpha(v) = \{e' : v \in e'\}$ for $v \in T$, and $\tau(e) = \{v' : v' \in e\} = e$ for $e \in A$. (4) $\theta = 2/3$. (5) $k = m$. Obviously, this construction can be done in polynomial time. We claim that $\mathrm{ACTL}(A, T, \theta, k)$ is true if and only if $\mathrm{EP}_3(G(V, E))$ is true.

If-part: If $\mathrm{EP}_3(G(V, E))$ is true, E is partitioned into $E_1, \ldots, E_m$ such that each E_i, where $1 \leq i \leq m$, generates a subgraph of G isomorphic to K_3. In other words, E_i has exactly 3 edges forming a complete graph on 3 vertices from V. We can construct the partition Π of $A = E$ such that $\Pi = \{E_1, \ldots, E_m\}$. For each $E_i \in \Pi$, supposing $E_i = \{e_1, e_2, e_3\}$, we have $\mathrm{LD}(E_i) = 6$.

While e_1, e_2, e_3 form a complete graph on 3 vertices, $e_1 \cup e_2 \cup e_3$ is the set of these 3 vertices. The load factor $\mathrm{LF}(E_i)$ for E_i is $6/(3 \times 3) = 2/3 = \theta$. Also, $|\Pi| = m = k$. Therefore, $\mathrm{ACTL}(A, T, \theta, k)$ is true.

Only-if-part: If $\mathrm{ACTL}(A, T, \theta, k)$ is true, there exists some partition Π of A such that $|\Pi| \leq k$ and $\mathrm{LF}(C) \geq 2/3$ for each $C \in \Pi$. For each $C \in \Pi$, every $e \in C$ is an attribute (edge) which is associated with a set of exactly 2 transactions (vertices). We have $\mathrm{LF}(C) = |C| \times 2/(|C| \times |\bigcup_{e \in C} C|) \geq 2/3$, which gives $|\bigcup_{e \in C} C| \leq 3$. Since C is a collection of distinct sets, each containing 2 elements, $|\bigcup_{e \in C} C| \leq 3$ implies $|C| \leq 3$ Based on (a) $|\Pi| \leq k$, (b) $|C| \leq 3$ for any $C \in \Pi$, and (c) $\sum_{C \in \Pi} |C| = |A| = 3k$, we can conclude Π has exactly k clusters and every $C \in \Pi$ contains exactly 3 elements. For each $C \in \Pi$, $\mathrm{LF}(C) = \mathrm{LD}(C)/(3 \times |\bigcup_{e \in C} C|) = 3 \times 2/(3 \times |\bigcup_{e \in C} C|) \geq 2/3$ because $|\bigcup_{e \in C} C| \leq 3$. Since C has exactly 3 distinct two-element-sets, $|\bigcup_{e \in C} C| \geq 3$. Hence, $|\bigcup_{e \in C} C| = 3$. Every $C \in \Pi$ contains exactly 3 edges (attributes) covering 3 vertices (transactions), which forms a subgraph of G isomorphic to K_3. Π is an EP_3 solution, which implies $\mathrm{EP}_3(G(V, E))$ is true.

Since ACTL is both in NP and NP-hard, it is NP-complete.

Appendix B. Queries on the Wikipedia Infobox Dataset

The queries on the Wikipedia dataset described in Sect. 9.4 are listed below. For each query, we first describe it in English. Then, we give its SQL statements for the schemas of the triple-store (TS), the vertical database (Vert), HoVer, and ACTL. The HoVer and ACTL SQLs involve the machine generated table and column names, which are prefixed by "t" and "a" respectively, followed by a number. The mappings between column names and attributes are listed in Table B.3.

Q1: Find the law schools with known rankings and their annual tuition fees.

TS:
```
SELECT subject_id, A.object_id AS ranking,
B.object_id AS fee FROM (SELECT subject_id,
object_id FROM triplets WHERE predicate_id =
23492) AS A LEFT OUTER JOIN (SELECT
subject_id, object_id FROM triplets WHERE
predicate_id = 23497) AS B USING (subject_id)
ORDER BY subject_id ASC;
```

Vert:
```
SELECT subject_id, A.object_id AS ranking,
B.object_id AS fee FROM p_23492 AS A LEFT
OUTER JOIN p_23497 AS B USING (subject_id)
ORDER BY subject_id ASC;
```

Table B.3: Column names in HoVer and ACTL schemas

Query	Attribute name	ACTL column name	HoVer column name
Q1	Law School#ranking	23492	23486
	Law School#annual tuition	23497	23488
Q2	Star Trek episode#name	1271	1282
	Star Trek episode#producer	21953	21953
	Star Trek episode#prod_num	1279	1279
Q3	Computer Hardware Printer#color	15286	15292
	Computer Hardware Printer#dpi	15292	15288
	Computer Hardware Printer#speed	15287	15291
	Computer Hardware Printer#slot	15297	15293
Q4	Laboratory#type	13493	13496
	Laboratory#budget	13497	13500
	Laboratory#website	13500	13492
	Laboratory#staff	13499	13494
	Laboratory#students	13498	13495
Q5	NBA Player#career_highlights	31595	31595
	NBA Player#height_ft	4243	4245
	NBA Player#weight_lbs	5887	5887
Q6	Officeholder#date_of_birth	28655	28657
	Officeholder#place_of_birth	28658	28655
	Officeholder#date_of_death	34972	34972
	Officeholder#place_of_death	34973	34973
Q7	London Bus#number	37023	37021
	London Bus#start	37013	37018
	London Bus#end	37025	37025
	London Bus#length	37017	37013
	London Bus#day	37019	37017
	London Bus#level	37021	37014
	London Bus#frequency	37024	37023
	London Bus#time	37014	37015
Q8	Library#location	13472	13474
	Library#website	13471	13470

HoVer:
```
SELECT subject, a23486 AS ranking, a23488 AS
fee FROM t2474 WHERE a23486 IS NOT NULL ORDER
BY subject ASC;
```

ACTL:
```
SELECT subject, a23492 AS ranking, a23497 AS
fee FROM t81008 WHERE a23492 IS NOT NULL
ORDER BY subject ASC;
```

Q2: What is the name and production number of the Start Trek episode of which the producer is Dawn Valazquez?

TS:
```
SELECT B.object_id AS name, C.object_id AS
prod_num FROM (SELECT subject_id FROM
triplets WHERE predicate_id = 21953 AND
object_id = 2921687) AS A LEFT OUTER JOIN
(SELECT subject_id, object_id FROM triplets
WHERE predicate_id = 1271) AS B USING
(subject_id) LEFT OUTER JOIN (SELECT
subject_id, object_id FROM triplets WHERE
predicate_id = 1279) AS C USING (subject_id)
ORDER BY name ASC;
```

Vert:
```
SELECT B.object_id AS name, C.object_id AS
prod_num FROM (SELECT subject_id FROM p_21953
WHERE object_id = 2921687) AS A LEFT OUTER
JOIN p_1271 AS B USING (subject_id) LEFT
OUTER JOIN p_1279 AS C USING (subject_id)
ORDER BY name ASC;
```

HoVer:
```
SELECT B.a1282 AS name, B.a1279 AS prod_num
FROM (SELECT subject FROM triplets WHERE
predicate = 21953 AND object = 2921687) AS A
LEFT OUTER JOIN t750 AS B USING (subject)
ORDER BY name ASC;
```

ACTL:
```
SELECT a1271 AS name, a1279 AS prod_num FROM
t81159 WHERE a21953 = 2921687 ORDER BY name ASC;
```

Q3: List the model numbers, speeds and numbers of slots of four-color printers wirh 600 dpi.

TS:
```
SELECT subject_id, C.object_id AS speed,
D.object_id AS slot FROM (SELECT subject_id
FROM triplets WHERE predicate_id = 15286 AND
object_id = 1000343) AS A INNER JOIN (SELECT
subject_id FROM triplets WHERE predicate_id =
15292 AND object_id = 1068340) AS B USING
(subject_id) LEFT OUTER JOIN (SELECT
subject_id, object_id FROM triplets WHERE
predicate_id = 15287) AS C USING (subject_id)
LEFT OUTER JOIN (SELECT subject_id, object_id
FROM triplets WHERE predicate_id = 15297) AS
D USING (subject_id) ORDER BY subject_id ASC;
```

Vert:
```
SELECT subject_id, C.object_id AS speed,
D.object_id AS slot FROM (SELECT subject_id
FROM p_15286 WHERE object_id = 1000343) AS A
INNER JOIN (SELECT subject_id FROM p_15292
WHERE object_id = 1068340) AS B USING
(subject_id) LEFT OUTER JOIN p_15287 AS C
USING (subject_id) LEFT OUTER JOIN p_15297 AS
D USING (subject_id) ORDER BY subject_id ASC;
```

HoVer:
```
SELECT subject, a15291 AS speed, a15293 AS
slot FROM t1626 WHERE a15292 = 1000343 AND
a15288 = 1068340 ORDER BY subject ASC;
```

ACTL:
```
SELECT subject, a15287 AS speed, a15297 AS
slot FROM t81388 WHERE a15286 = 1000343 AND
a15292 = 1068340 ORDER BY subject ASC;
```

Q4: List all the laboratories with any of the following information known: types, budgets, websites, staff counts and student counts.

TS:
```
SELECT subject_id, A.object_id AS type,
B.object_id AS budget, C.object_id AS
website, D.object_id AS staff_count,
E.object_id AS student_count FROM (SELECT
subject_id, object_id FROM triplets WHERE
predicate_id = 13493) AS A FULL OUTER JOIN
(SELECT subject_id, object_id FROM triplets
WHERE predicate_id = 13497) AS B USING
(subject_id) FULL OUTER JOIN (SELECT
subject_id, object_id FROM triplets WHERE
predicate_id = 13500) AS C USING (subject_id)
FULL OUTER JOIN (SELECT subject_id, object_id
FROM triplets WHERE predicate_id = 13499) AS
D USING (subject_id) FULL OUTER JOIN (SELECT
subject_id, object_id FROM triplets WHERE
predicate_id = 13498) AS E USING (subject_id)
ORDER BY subject_id ASC;
```

Vert:
```
SELECT subject_id, A.object_id AS type,
B.object_id AS budget, C.object_id AS
website, D.object_id AS staff_count,
E.object_id AS student_count FROM p_13493 AS
A FULL OUTER JOIN p_13497 AS B USING
(subject_id) FULL OUTER JOIN p_13500 AS C
USING (subject_id) FULL OUTER JOIN p_13499 AS
D USING (subject_id) FULL OUTER JOIN p_13498
AS E USING (subject_id) ORDER BY subject_id ASC;
```

HoVer:
```
SELECT subject, a13496 AS type, a13500 AS
budget, a13492 AS website, a13494 AS
staff_count, a13495 AS student_count FROM
t4105 WHERE a13496 IS NOT NULL OR a13500 IS
NOT NULL OR a13492 IS NOT NULL OR a13494 IS
NOT NULL OR a13495 IS NOT NULL ORDER BY
subject ASC;
```

ACTL:
```
SELECT subject, a13493 AS type, a13497 AS
budget, a13500 AS website, a13499 AS
staff_count, a13498 AS student_count FROM
t80539 WHERE a13493 IS NOT NULL OR a13497 IS
NOT NULL OR a13500 IS NOT NULL OR a13499 IS
NOT NULL OR a13498 IS NOT NULL ORDER BY
subject ASC;
```

Q5: Which NBA players have career highlights listed? State also their weights and heights.

TS:
```
SELECT subject_id, A.object_id AS
career_highlight, B.object_id AS height,
C.object_id AS weight FROM (SELECT
subject_id, object_id FROM triplets WHERE
predicate_id = 31595) AS A LEFT OUTER JOIN
(SELECT subject_id, object_id FROM triplets
WHERE predicate_id = 4243) AS B USING
(subject_id) LEFT OUTER JOIN (SELECT
subject_id, object_id FROM triplets WHERE
predicate_id = 5887) AS C USING (subject_id)
ORDER BY subject_id ASC;
```

Vert:
```
SELECT subject_id, A.object_id AS
career_highlight, B.object_id AS height,
C.object_id AS weight FROM p_31595 AS A LEFT
OUTER JOIN p_4243 AS B USING (subject_id)
LEFT OUTER JOIN p_5887 AS C USING
(subject_id) ORDER BY subject_id ASC;
```

HoVer:
```
SELECT subject, A.object AS career_highlight,
B.a4245 AS height, C.object AS weight FROM
(SELECT subject, object FROM triplets WHERE
predicate = 31595) AS A LEFT OUTER JOIN
(SELECT subject, a4245 FROM t152) AS B USING
(subject) LEFT OUTER JOIN (SELECT subject,
object FROM triplets WHERE predicate = 5887)
AS C USING (subject) ORDER BY subject ASC;
```

ACTL:
```
SELECT subject, A.a31595 AS career_highlight,
B.a4243 AS height, B.a5887 AS weight FROM
(SELECT subject, a31595 FROM t81315 WHERE
a31595 IS NOT NULL) AS A LEFT OUTER JOIN
(SELECT subject, a4243, a5887 FROM t81694) AS
B USING (subject) ORDER BY subject ASC;
```

Q6: List all the officeholders with known date of birth, birthplace, date of death or death place

TS:
```
SELECT subject_id, A.object_id AS birth_date,
B.object_id AS birth_place, C.object_id AS
death_date, D.object_id AS death_place FROM
(SELECT subject_id, object_id FROM triplets
WHERE predicate_id = 28655) AS A FULL OUTER
JOIN (SELECT subject_id, object_id FROM
triplets WHERE predicate_id = 28658) AS B
USING (subject_id) FULL OUTER JOIN (SELECT
subject_id, object_id FROM triplets WHERE
predicate_id = 34972) AS C USING (subject_id)
FULL OUTER JOIN (SELECT subject_id, object_id
FROM triplets WHERE predicate_id = 34973) AS
D USING (subject_id) ORDER BY subject_id ASC;
```

Vert:
```
SELECT subject_id, A.object_id AS birth_date,
B.object_id AS birth_place, C.object_id AS
death_date, D.object_id AS death_place FROM
p_28655 AS A FULL OUTER JOIN p_28658 AS B
USING (subject_id) FULL OUTER JOIN p_34972 AS
C USING (subject_id) FULL OUTER JOIN p_34973
AS D USING (subject_id) ORDER BY subject_id ASC;
```

HoVer:
```
SELECT subject, A.a28657 AS birth_date,
A.a28655 AS birth_place, B.a34972 AS
death_date, B.a34973 AS death_place FROM
(SELECT subject, a28657, a28655 FROM t8172
WHERE a28657 IS NOT NULL OR a28655 IS NOT
NULL) AS A FULL OUTER JOIN (SELECT subject,
a34972, a34973 FROM t14873 WHERE a34972 IS
NOT NULL OR a34973 IS NOT NULL) AS B USING
(subject) ORDER BY subject ASC;
```

ACTL:
```
SELECT subject, A.a28655 AS birth_date,
A.a28658 AS birth_place, B.a34972 AS
death_date, B.a34973 AS death_place FROM
(SELECT subject, a28655, a28658 FROM t80008
WHERE a28655 IS NOT NULL OR a28658 IS NOT
NULL) AS A FULL OUTER JOIN (SELECT subject,
a34972, a34973 FROM t61116 WHERE a34972 IS
NOT NULL OR a34973 IS NOT NULL) AS B USING
(subject) ORDER BY subject ASC;
```

Q7: Find the bus routes to or from Heathrow

Airport. Show the route numbers, starting point, ending point, journey length, service time during the day, service level, frequency and journey time.

TS:
```
SELECT A.object_id AS bus_num, B.object_id AS
start, C.object_id AS end, D.object_id AS
length, E.object_id AS day, F.object_id AS
level, G.object_id AS frequency, H.object_id
AS time FROM (SELECT subject_id, object_id
FROM triplets WHERE predicate_id = 37013) AS
B FULL OUTER JOIN (SELECT subject_id,
object_id FROM triplets WHERE predicate_id =
37025) AS C USING (subject_id) LEFT OUTER
JOIN (SELECT subject_id, object_id FROM
triplets WHERE predicate_id = 37023) AS A
USING (subject_id) LEFT OUTER JOIN (SELECT
subject_id, object_id FROM triplets WHERE
predicate_id = 37017) AS D USING (subject_id)
LEFT OUTER JOIN (SELECT subject_id, object_id
FROM triplets WHERE predicate_id = 37019) AS
E USING (subject_id) LEFT OUTER JOIN (SELECT
subject_id, object_id FROM triplets WHERE
predicate_id = 37021) AS F USING (subject_id)
LEFT OUTER JOIN (SELECT subject_id, object_id
FROM triplets WHERE predicate_id = 37024) AS
G USING (subject_id) LEFT OUTER JOIN (SELECT
subject_id, object_id FROM triplets WHERE
predicate_id = 37014) AS H USING (subject_id)
WHERE (B.object_id = 2980136 OR C.object_id =
2980136) ORDER BY bus_num ASC;
```

Vert:
```
SELECT A.object_id AS bus_num, B.object_id AS
start, C.object_id AS end, D.object_id AS
length, E.object_id AS day, F.object_id AS
level, G.object_id AS frequency, H.object_id
AS time FROM p_37013 AS B FULL OUTER JOIN
p_37025 AS C USING (subject_id) LEFT OUTER
JOIN p_37023 AS A USING (subject_id) LEFT
OUTER JOIN p_37017 AS D USING (subject_id)
LEFT OUTER JOIN p_37019 AS E USING
(subject_id) LEFT OUTER JOIN p_37021 AS F
USING (subject_id) LEFT OUTER JOIN p_37024 AS
G USING (subject_id) LEFT OUTER JOIN p_37014
AS H USING (subject_id) WHERE (B.object_id =
2980136 OR C.object_id = 2980136) ORDER BY
bus_num ASC;
```

HoVer:
```
SELECT A.a37021 AS bus_num, B.object AS
start, C.object AS end, A.a37013 AS length,
D.object AS day, A.a37014 AS level, A.a37023
AS frequency, A.a37015 AS time FROM (SELECT
subject, object FROM triplets WHERE predicate
= 37018) AS B FULL OUTER JOIN (SELECT
subject, object FROM triplets WHERE predicate
= 37025) AS C USING (subject) LEFT OUTER JOIN
(SELECT subject, a37021, a37013, a37014,
a37023, a37015 FROM t443) AS A USING
(subject) LEFT OUTER JOIN (SELECT subject,
object FROM triplets WHERE predicate = 37017)
AS D USING (subject) WHERE (B.object =
2980136 OR C.object = 2980136) ORDER BY
bus_num ASC;
```

ACTL:
```
SELECT A.a37023 AS bus_num, B.object AS
start, C.object AS end, A.a37017 AS length,
D.object AS day, A.a37021 AS level, A.a37024
AS frequency, A.a37014 AS time FROM (SELECT
subject, object FROM triplets WHERE predicate
= 37013) AS B FULL OUTER JOIN (SELECT
subject, object FROM triplets WHERE predicate
= 37025) AS C USING (subject) LEFT OUTER JOIN
(SELECT subject, a37023, a37017, a37021,
a37024, a37014 FROM t80764) AS A USING
(subject) LEFT OUTER JOIN (SELECT subject,
object FROM triplets WHERE predicate = 37019)
AS D USING (subject) WHERE (B.object =
2980136 OR C.object = 2980136) ORDER BY
bus_num ASC;
```

Q8: Find all the libraries with known locations and websites.

TS:
```
SELECT subject_id, A.object_id AS location,
B.object_id AS website FROM (SELECT
subject_id, object_id FROM triplets WHERE
predicate_id = 13472) AS A INNER JOIN (SELECT
subject_id, object_id FROM triplets WHERE
predicate_id = 13471) AS B USING
(subject_id)ORDER BY subject_id ASC;
```

Vert:
```
SELECT subject_id, A.object_id AS location,
B.object_id AS website FROM p_13472 AS A
INNER JOIN p_13471 AS B USING (subject_id)
ORDER BY subject_id ASC;
```

HoVer:
```
SELECT subject, A.object AS location,
B.a13470 AS website FROM (SELECT subject,
object FROM triplets WHERE predicate = 13474)
AS A INNER JOIN (SELECT subject, a13470 FROM
t1378 WHERE a13470 IS NOT NULL) AS B USING
(subject) ORDER BY subject ASC;
```

ACTL:
```
SELECT subject, A.object AS location,
B.a13471 AS website FROM (SELECT subject,
object FROM triplets WHERE predicate = 13472)
AS A INNER JOIN (SELECT subject, a13471 FROM
t80919 WHERE a13471 IS NOT NULL) AS B USING
(subject) ORDER BY subject ASC;
```