Structural indexing for accelerated join-processing in relational databases
Abstract

An increasing amount of data is generated and stored every day. This phenomenon called ‘Big Data’ requires innovative solutions for data storage and query processing. In contrast with the large amount of data being stored in databases, the retrieved information is often small and may be derived from a fraction of the data. Retrieving valuable information from a large collection of data can be like ‘finding the needle in a haystack’.

Indexing techniques enable fast random access to the desired information. Data can directly be located by ordering or hashing records on their attributes. However, sometimes we want to retrieve records based on their relational structure. Structural indexing has been used in the past for accelerating graph databases and RDF-databases to solve this problem and it may also be suitable for relational databases.

To our knowledge, this report provides the first known empirical evaluation of structural indexing in relational databases. We show how structural similarity can be used in relational databases to accelerate query processing. By using $k$-bisimulation partitioning, we group together bisimilar records to obtain a compact representation of the data structure that serves as a structural index. We accelerate semijoin and antijoin queries by several orders of magnitude by pruning information from the index to select or eliminate tuples that do participate in the join.

**Keywords:** relational database, bisimulation, guarded structural indexing, conjunctive queries, guarded fragment queries, join processing
Preface

This thesis is a result of my graduation project within the Web Engineering research group at the Eindhoven University of Technology. The project was done in collaboration with the National University of Singapore, Université libre de Bruxelles and TU Delft.

I would like to thank my supervisor George Fletcher for guiding me through this project. Further, I like to thank Stijn Vansummeren and Chee Yong Chan from Bruxelles and Singapore for their ideas and constructive feedback on thesis drafts. Also, I would like to thank Yongming Luo for his support in adapting his software to my project. Last but not least, I like to thank my parents, family and friends for their support during my study.

Erik Agterdenbos
September 2015
Contents

1 Introduction  6
  1.1 Motivation  6
  1.2 Problem definition  6
  1.3 Objective and methodology  7
  1.4 Results  7
  1.5 Outline  7

2 Background  8
  2.1 Positioning  8
  2.2 Stakeholders in this project  8
  2.3 Related work  8
  2.4 Definitions  9

3 Our approach  10
  3.1 Graph representation  10
  3.2 Notation  11
  3.3 Partitioning and graph reduction  11
  3.4 Structural index storage  12
    3.4.1 Partitioning storage  12
    3.4.2 Reduced graph storage  12
    3.4.3 Alternative partitioning storage  12
  3.5 Query acceleration  13
    3.5.1 Semi-join and anti-join acceleration  13
    3.5.2 Equijoin acceleration  15
    3.5.3 Value-based selection conditions  16
  3.6 Hybrid indexing  16
  3.7 Workload-driven structural indexing  17
    3.7.1 k-Bisimulation structural indexing on selected relations  17
    3.7.2 Relation partitioning on query result  17
    3.7.3 DAG-based structural indexing  17

4 Experimental study  20
  4.1 Pruning power evaluation for structural indexing  20
    4.1.1 Data  20
    4.1.2 Running time evaluation for single joins  21
    4.1.3 Discussion  23
  4.2 Implementation for k-Bisimulation structural indexing  24
  4.3 k-Bisimulation structural indexing for TPC-H  25
    4.3.1 Data  25
    4.3.2 Experimental design  27
    4.3.3 Reduced graph analysis  27
    4.3.4 Structural Index Size Analysis  30
    4.3.5 Query Performance Analysis  30
    4.3.6 Discussion  34
  4.4 Hybrid indexing on TPC-H  36
    4.4.1 Implementation  36
    4.4.2 Reduced graph analysis  36
## CONTENTS

| 4.4.3  | Query performance analysis | 37 |
| 4.4.4  | Discussion               | 37 |
| 4.5    | k-Bisimulation structural indexing for DBLP | 37 |
| 4.5.1  | Data                     | 37 |
| 4.5.2  | Experimental design      | 38 |
| 4.5.3  | Reduced graph analysis   | 38 |
| 4.5.4  | Structural index size analysis | 40 |
| 4.5.5  | Query performance analysis | 40 |
| 4.5.6  | Discussion               | 44 |
| 4.6    | Workload-driven structural indexing | 44 |
| 4.6.1  | Relation partitioning on query result | 44 |
| 4.6.2  | DAG-based structural indexing | 45 |

5 Conclusions 48

| 5.1    | Contributions           | 48 |
| 5.2    | Limitations             | 49 |
| 5.3    | Future work             | 49 |

A Structural Index schema for PostgreSQL 52

B TPC-H Schema for PostgreSQL 53

C TPC-H Queries 56

| C.1  | Freely Acyclic Conjunctive Queries | 56 |
| C.2  | Guarded Fragment Queries | 60 |
| C.3  | Value-based Selective Queries | 64 |
| C.4  | Hybrid Queries | 65 |

D DBLP Schema for PostgreSQL 68

E DBLP Queries 71

| E.1  | Freely Acyclic Conjunctive Queries | 71 |
| E.2  | Guarded Fragment Queries | 77 |
Chapter 1

Introduction

1.1 Motivation

Indices in current relational databases use value-based indexing. This means that records are sorted on the value of one or more attributes to enable fast random lookups [16]. Selective queries that retrieve a small subset of the data can benefit from these indices. A selection operator eliminates records where the values do not satisfy the selection predicate. However, some queries are not selective by value but selective by structure. One or more joins are required to determine which records must be returned. These queries cannot benefit from value-based indexing. When the query is not selective by value, the number of disk seeks may become very large and indexed nested loop-joins become slow. Multiple table scans are required to retrieve all tuples in the query result. This makes the join a relatively expensive operation in query processing. Moreover, join-intensive query runtime performance degrades when the data set grows [13].

For graph databases and RDF databases, so-called structural indices are sometimes used to answer queries. In contrast to value-based indices, these structural indices group together structurally similar nodes to obtain a more compact representation of the graph. Bisimilarity, a well studied formalization of similarity is typically used. Some queries can directly be answered on the index (covering index) without accessing the original database. Otherwise, the structural index is used for pruning to answer the query on the original graph with higher efficiency [12]. These techniques reduce the negative impact of joins on query runtime performance when the dataset grows.

To our knowledge, structural indices have never been used before in relational databases. A recent formal study from Picalausa et al. [12] showed that structural indices can be extended from graph databases to the field of relational databases. Their paper characterizes queries that can benefit from accelerated join processing by using structural indices. Further, it shows how query guardedness can be maintained when k-bisimilarity, an approximation of full bisimilarity, is used to build the structural index.

1.2 Problem definition

Graph databases are often more efficient than relational databases for join-intensive queries [13]. With “join-intensive queries”, we mean queries that contain multiple join operations. Join-intensive queries on relational databases can become very slow, in particular when no selection conditions are used. Selection conditions can restrict the attributes to given values like “ID = 42”, which can make the joins much faster because only some tuples participate in the joins.

This observation leads to the following problem formulation: “Relational databases are slow at processing join intensive queries without value-based selection conditions on large data sets.” The question is whether structural indexing can increase the efficiency of join processing. In our approach to solve this problem, we focus on a subset of join operations: joins that have only equality predicates ‘=’ in the join condition, also called equi-joins. Queries that have ‘<’ or ‘>’ join conditions may be slow as well but these cannot be directly answered by a graph database.
1.3 Objective and methodology

Our project consists of mainly engineering and experimentation. The main goal is to provide an empirical evaluation for the claim from Picalausa et al. that structural indexing can be extended from graph databases to arbitrary relational databases. The goal is to accelerate queries on a relational database. We want to measure the acceleration of join intensive queries by using structural indexing.

The main task of is to build a structural index for a particular relational database and then use this index to evaluate equi-join queries. This includes the creation of a bisimulation partitioning for the structural index, design of data structures to store the index and partitioning and finally an evaluation of the query types that can benefit from the index.

There are some challenges when a structural index is used for relational databases. Instead of grouping together structurally similar nodes, we need to group together structurally similar tuples. Several external memory bisimulation partitioning algorithms exists for graphs [3, 8]. To use these partitioning algorithms for relational databases, we first need to represent relational databases as graphs. Further, queries must be compiled to a query execution plan that exploits graph structure.

1.4 Results

We provide an approach to create and use a structural index for an arbitrary relational database. Our structural index can deliver faster query processing for several types of queries while using a smaller amount of disk space than traditional B-tree indexing. The relational database can be represented as a graph consisting of nodes and edges by using foreign keys. Then, we apply an existing $k$-bisimulation partitioning algorithm to derive a structural index for the relational database. We visualize the graph structure of the structural index and discuss which information can be obtained from it. We also evaluate some approaches to build a structural index specifically to a particular workload.

To accelerate join query processing, we demonstrate how semijoin and antijoin queries can be accelerated by several orders of magnitude. We can accelerate semijoins or antijoins either by completely evaluating these joins on the structural index, or by pruning information from the structural index.

Some queries have both structural and nested value-based selection conditions. These queries cannot be answered by a structural index. We show how we can build a hybrid index that contains both structural information and some attribute values to answer these type of queries. Our experiments showed that an acceleration is still possible for these type of queries.

We have also evaluated some approaches that did not give a significant query acceleration. Our approach for tuple generating equijoin queries gave only a very small improvement or even slowed down the query processing.

1.5 Outline

Chapter 2 introduces the background of this project, possible applications and related work. Chapter 3 describes our approach. In Chapter 4 we describe the experimental evaluation and results. Finally, we derive a conclusion in Chapter 5.
Chapter 2

Background

2.1 Positioning

The project is part of a joint research project with Chee Yong Chan at National University of Singapore, Stijn Vansummeren at Université libre de Bruxelles, Jan Hidders at TU Delft and George Fletcher at TU Eindhoven.

2.2 Stakeholders in this project

Relational databases are widely used in many organizations. Join operations are very common in queries on relational databases. Previous research showed that 75% of the evaluated SPARQL queries in RDF databases have characteristics that may exploit structural indices [9]. Therefore, it is expected that also many relational databases may benefit from structural indices. Even if the performance improvement is relatively modest, any progress is still welcome for many organizations.

Graph databases are often used to store social relationships because of the join-intensive queries. A combination of a graph database and a relational database is used for applications with social networks. When the join performance properties of graph databases can be achieved in relational databases, future applications that contain a social network structure may only need one type of database that should combine the strengths of graph databases and relational databases.

2.3 Related work


A literature study for path-based XML queries was conducted by Wong et al. [20]. The path index of XML-databases can be seen as a structural index with a tree structure. Because every XML document has a root, also the queries are expected to start from the root. In practice, it may be the case that only the structure of a sub tree is known. In that case, a different index is required to answer these queries efficiently. Most existing indices are also not optimized for disk-based access. For this reason, there are still many open issues.

A literature study about storing and indexing of RDF data sets was performed by Luo et al. [9]. Section 4.2 provides an overview of the first use of structural indexing in graph data databases and how it moved to XML and finally to RDF databases by Tran and Ladwig [19]. In their solution, the edge label is removed from the triple pattern to create a graph structure. This allowed to use structural indices that were designed for graph databases, but removes the possibility to query for these labels. In 2011, Zou et al. [21] proposed gStore, an RDF database that can answer exact SPARQL queries from a structural index and can process index updates as well.

In 2012, Hellings et al. [3] introduced the first efficient external memory bisimulation algorithm for directed acyclic graphs. Luo et al. [8] generalized this approach by presenting an external memory k-bisimulation algorithm for directed graphs that does not rely on the acyclic structure. In 2013, Picalausa [11] adapted several bisimulation partitioning algorithms to RDF databases as part of his PhD thesis. Following this approach, it may be possible to adapt these algorithms to relational databases as well.
The main resource for this project is “Principles of Guarded Structural Indexing” from [12]. This formal study introduces the use of structural indices in relational databases. Only acyclic conjunctive queries are evaluated in this paper. Their conclusion is that structural indexing can also be used in relational databases to answer acyclic conjunctive queries. An experimental evaluation of their approach should still be performed.

Spiegel and Polyzotis [17] presented a method to efficiently approximate SUM and COUNT SQL queries by using a graph structure. Their contributions may also be relevant for this project because the authors can provide approximate answers for some complex equi-join queries using structural statistics without processing the joins.

2.4 Definitions

Bisimulation and simulation are two different notions of structural similarity in graphs [14]. Guarded bisimulation and guarded simulation extend these notions from graphs to relational databases [12]. Instead of (bi)similar nodes in a graph, guarded (bi)simulation defines when relational tuples in a relational database have a similar structure and can be called guarded (bi)similar. These notions can be used for structural indexing relational database. Each notion has its own characterization of queries that can be answered on structural indices that are built using a particular notion. We will explain two characterizations of queries for relational databases: freely acyclic conjunctive queries and guarded fragment queries.

Freely acyclic conjunctive queries (FACQ) are queries that can be expressed using conjunction, disjunction and existential quantification. Negation and universal quantification, two other elements of first order logic, are not allowed in FACQ queries. Furthermore, freely acyclic means that no cycles are allowed when the query head and body are combined, which means that the join program can be represented by a tree. The database, however, may contain cycles. FACQ queries are called ‘invariant’ under guarded simulation and guarded bisimulation [12]. This means that the query cannot distinguish relational tuples that are guarded similar or guarded bisimilar. As a result, FACQ queries can be answered by structural indices built using guarded simulation or guarded bisimulation.

Guarded fragment queries are an extension to FACQ queries. These queries also allow full first-order logic and can be expressed using conjunction, disjunction and existential quantification, but also negation and universal quantification can be used. Guarded fragment queries are invariant under guarded bisimulation [12], but not under guarded simulation. Guarded bisimulation is a stronger notion of structural similarity than guarded simulation. Two relational tuples that are not guarded bisimilar can still be guarded similar.

$k$-Bisimulation is a localized approximation of bisimulation and can be used as an alternative for normal bisimulation. It considers local topological features of nodes within a neighborhood of distance $k$ [8]. By choosing a value for $k$, we can control the granularity of the bisimulation. FACQ queries and guarded fragment queries with join tree height $h$ are invariant under guarded $k$-bisimulation if $h \leq k$ [12]. We can ensure the invariance of our queries by choosing a sufficiently large value for $k$. 

Chapter 3

Our approach

Picalausa applied guarded bisimulation reduction for RDF databases as part of his PhD thesis. The idea in our work is to apply the same guarded bisimulation reduction technique on relational databases. Picalausa et al. explained in [12] how guarded bisimulation can be performed on relational databases.

In our approach, we use the external memory $k$-bisimulation partitioning algorithm for graphs that was presented by Luo et al. [8]. Algorithms for the construction and maintenance of the $k$-bisimulation partitioning have been implemented as part of their work. An external memory algorithm is used because the size of the dataset may grow beyond the size of the internal memory. An implementation that is not optimized for external memory usage would slow down the $k$-bisimulation computation.

Our approach to build and use a $k$-bisimulation structural index can be summarized as follows: We convert the relational multi-arity tuples to graph edges with respect to guardedness. Then we apply the existing $k$-bisimulation partitioning algorithm to group together $k$-bisimilar nodes, which results in a smaller graph that serves as an index. Finally, we (partially) evaluate queries using the index. The details in each step will be given in the following sections.

At the end of the chapter, we will give an approach to embed attribute value information in structural indices to obtain a hybrid structural index. A hybrid structural index can answer queries that are both selective by value and structure. Further, we will present several approaches to build structural indices for a specific workload.

3.1 Graph representation

In relational databases, joins can be performed on any set of attribute pairs having equal data type. However, joins on foreign keys are the most common. We will focus on joins for which the join condition is such that primary key attributes are equal to the foreign key attributes. Foreign key constraints are typically added to data definition language to ensure referential integrity of the data [16]. We can also use these constraints to identify candidates for equi-join queries.

The structure of the data in a relational database can be represented by a graph that contains both node labels and edge labels. The construction of a graphs is as follows. For each tuple in the database, we create a node. The relation name and all primary key values are combined to derive a unique node identifier, called node id. The relation name will be used for the node label. During bisimulation partitioning we ensure that nodes with distinct node labels are assigned to different partition blocks. For each foreign key in the database, we create forward edges and backward edges. We draw a ‘forward’ edge from the referencing node to the referenced node. Similarly, we draw a ‘backward’ edge from the referenced node to the referencing node. The foreign key name is used as edge label prefixed with fw or bw for the direction.

The direction of the edge influences join capabilities and bisimulation results. Undirected graphs may be preferred but the existing external memory $k$-bisimulation algorithms are only for directed graphs. Therefore, we create directional edges in both directions for each foreign key relationship.

We will now give an example of a small relational database represented as a graph. Consider a relation order(ID, status, price, customer_id), a relation customer(ID, name, town) and a foreign key called order_cid_fkey from order.customer_id to customer.ID. Figure 3.1 gives an example dataset for this schema and Figure 3.2 shows the graph that corresponds with this dataset.
3.2 Notation

We combine forward and backward edges into one bidirectional edge notation to improve readability and allow node ranking. Figure 3.3 shows the schema notation, the tuple notation and partitioning notation. Edges are drawn from the primary key (open arrow) to the foreign key (closed arrow). The schema in graph notation can be recognized by rectangular nodes. Their relation names are used for node identifiers and foreign key names are used for edge labeling.

Tuples are represented by ellipse nodes. Their relation name and primary key are used for node labeling. Foreign key names are used for edge labeling.

Rounded rectangles are used to draw partition blocks. Partition blocks are groups of tuples with a similar structure: these tuples belong to the same bisimulation equivalence class. The nodes contain the relation name, partition block identifier and the size. The size denotes number of tuples that are assigned to the partition block. The partitioning notation contains also edges. These edges between partition blocks represent groups of structural similar edges in the graph notation for tuples. Again, foreign key names are used for edge labeling. The partition blocks and edges between those blocks form a ‘reduced graph’: a more compact graph representation of the original tuples in graph representation.

3.3 Partitioning and graph reduction

We apply $k$-bisimulation partitioning on the graph that represents the structure of the database. Our input graph contains node labels and edge labels. Node labels are used to create an initial partitioning
of nodes on their relation names. Edge labels are used for further partitioning of the nodes during the $k$-bisimulation iterations. The partitioning finishes after a chosen number of iterations and is stored.

The $k$-bisimulation partitioning algorithm typically only assigns nodes to partition blocks. We have to construct the ‘reduced graph’ ourselves. Based on the partitioning and original graph, we can create a ‘reduced graph’ using the notation in Figure 3.3. First, we iterate over all edges and substitute the source and destination identifiers by the partition identifier of the referred nodes. Then we sort all edges, remove the duplicates and obtain a set of reduced edges. Second, we sort all nodes on their partition identifiers and count the number of nodes in each partition block. The result is a set of partition blocks with sizes and reduced edges which form a reduced graph. An example of a reduced graph, also called partitioning, is shown in Figure 3.3.

3.4 Structural index storage

The structural index consists of two parts that are stored separately:

**Reduced graph** a compact graph representation of the original database structure

**Partitioning** contains the mapping between tuples and partition blocks (nodes in the reduced graph)

We will describe their storage in the following subsections.

3.4.1 Partitioning storage

Each tuple in the original database is assigned to a partition block. To store this partitioning, we add a `block_id` attribute to each relation in the original schema. For each tuple, the value of this `block_id` attribute refers to a node in the reduced graph.

3.4.2 Reduced graph storage

We will now describe how the reduced graph is stored. Nodes and edges of the reduced graph are stored in a ‘structural index’ schema in the relational database. The reduced graph is a simplification of the graph representation of the original database, however, we cannot store the reduced graph in the original schema. The reduced graph must be stored separately. The reason for this is as follows: Foreign key constraints in the original schema represent one-to-one or one-to-many relationships. During graph reduction/partitioning, these relations become many-to-many relationships. As a result, the nodes and edges cannot be stored as tuples in the relations from which they originate.

The different schema for the reduced graph will be as follows. For each relation in the original schema, a ‘node’ type relation is created in the structural index schema. For each foreign key in the original schema, an ‘edge’ relation is created in the structural index schema. Node relations are of the form `node(Id, size)` and edge relations have the form `edge(from_id, to_id)`. The `size` attribute denotes the number of tuples that belong to a node.

We also add traditional indices for the reduced graph. For node relations, a B-tree is created on the primary key. Only forward edges are stored in the edge relations. Two B-trees are created for edge relations: `edge(to_id, from_id)` and `edge(from_id, to_id)`. This allows fast lookup from both the source and destination side.

3.4.3 Alternative partitioning storage

An alternative solution to store the logical node partitioning would be to use horizontal table partitioning. When horizontal table partitioning is used, the Database Management System (DBMS) distributes records to different files based on one or more chosen attributes, in this case the `block_id` attribute. All records that belong to the same partitioning block id will end up in the same file on the disk. This alternative may have some advantages. Firstly, the partition block id can be obtained from the location of the tuple. Therefore, the additional `block_id` attribute is no longer required. As a result, storing the partitioning requires less disk space when the number of partition blocks is small. Further, we can use partition pruning which means that partition blocks that do not contain matching identifiers do not have to be scanned. The disadvantage of this alternative is that records must move to another physical location when the `block_id` is updated. Most database management systems implement multiversion concurrency control (MVCC) to support transactional consistency. An important characteristic of MVCC is
that records move to a different physical location in case of updates. The old record will be deleted when
the transaction is finished. In that case, the physical move in case of updates is no longer a disadvantage.
A main disadvantage is that a very large number of partition blocks leads to storage overheads and slow
query planner performance.

3.5 Query acceleration

The goal is to have a structural index that is smaller than the original database, but still represents
the original structure. To speedup queries there are two options:

1. Directly answer queries on the index.
2. Accelerate queries using the original database by pruning information from the index.

When the query can directly be answered on the index, which is typically smaller than the original
database, it is likely that the number of I/O operations is also smaller. Therefore, we can expect higher
query performance.

When only parts of the query can be answered on the index, this can also improve performance. One
idea is that the index is used to eliminate a significant amount of records from the original database that
cannot be part of the output result. As a result, only a subset of the relations have to be accessed to
answer the query. Alternatively, relations are read but only parts have to be kept in memory to answer
the query.

Another idea is to use the index for clustering the original database in such a way that records with
a similar structure are stored at the same place to improve spatial locality of the query. This can be
achieved by physical horizontal table partitioning or clustering indexes.

The following subsections discuss several types of queries and potential acceleration possibilities.

3.5.1 Semijoin and antijoin acceleration

To speedup queries that contain semijoin operations, we substitute semijoin-(sub)trees by index semijoin-
(sub)trees. The idea is that the index is smaller than the original database and therefore also results
in faster semijoin processing. In relational databases, equi-joins can be performed on any pair attribute
sets that have the same type, but in our graph representation, we only create edges for foreign key
relationships. As a result, we can only accelerate equi-joins on a set of attribute pairs for which there
exists a foreign key with matching set or subset of attribute pairs.

Semijoins ($\bowtie$) and antijoins ($\bar{\bowtie}$) retrieve a subset of the records from one relation. Let $r$ and $s$ be
relations and let $R$ be the set of attributes in the schema of $r$, then the semijoin and antijoin operations
are defined as follows:

$$
\begin{align*}
    r \bowtie \theta s & \overset{\text{def}}{=} \Pi_R(r \bowtie \theta s) \\
    r \bar{\bowtie} \theta s & \overset{\text{def}}{=} r - \Pi_R(r \bowtie \theta s).
\end{align*}
$$

The semijoin selects those tuples $r_i$ in $r$ for which there are one or more tuples $s_j$ in $s$ such that the
join predicate $\theta$ holds. Similarly, antijoin selects those tuples $r_i$ in $r$ for which there is no tuple $s_j$ in $s$
such that the join predicate $\theta$ holds. We have $r \bowtie s \subseteq r$ and $r \bar{\bowtie} s \subseteq r$, so relation $s$ is not in the output
resultset, but is still accessed to determine which tuples must be retrieved. If $s$ is very large, this may
impact query performance.

To speedup query execution, we can query the structural index to determine which tuples in the
original database must be returned. The semijoin is performed on the index and the resulting partitioning
block IDs are used to select the required tuples without accessing the relation on the right side of the
semijoin.

We can substitute the semijoin operator as follows. First we look at the join condition and find
the foreign key constraint that matches the particular join predicate. Then, in the structural index
schema, which stores the reduced graph, we select the relation with the name that matches the foreign
key constraint name. Finally, we replace the original semijoin by a semijoin on the selected relation in the
structural index schema. The join condition is replaced as well. Attributes from the relation on the left
hand side of the semijoin operator are replaced with the block_id and the attributes from the relation
on right hand side of the semijoin operator are replaced with either from_id or to_id. This decision is
made as follows: primary key attributes in the join condition are replaced with from\_id and foreign key attributes are replaced with to\_id.

Suppose we have relations \( r(a, b, c) \) and \( s(d, e, f) \). Let \( s.f\_key \) be a foreign key from \( s(f) \) to \( r(a) \). Then we can accelerate the query \( r \bowtie_{r.a=s.f} s \) by substituting the semijoin operator by a selection operator:

\[
\begin{align*}
  r \bowtie_{r.a=s.f} s & := \sigma_\varphi(r) \\
  \text{where } \varphi & = \exists t_i \in \text{index.s.f\_key}(t_i, \text{to\_id} = r.\text{block\_id})
\end{align*}
\]

Similarly, antijoins can be substituted as follows:

\[
\begin{align*}
  r \bowtie_{r.a=s.f} s & := \sigma_\varphi(r) \\
  \text{where } \varphi & = \neg\exists t_i \in \text{index.s.f\_key}(t_i, \text{to\_id} = r.\text{block\_id})
\end{align*}
\]

**Subtree substitution algorithm**

We can substitute nested semijoins and/or antijoins as well. The first step is to draw a join tree of the semijoin and antijoin operations. The relation that contains the retrieved tuples becomes the root of the tree. Each semijoin or antijoin operation adds a child to the tree.

In the second step, we select the largest possible subtrees from the join tree. If we have an \( k \)-bisimulation structural index, we can select subtrees of height \( h \leq k \) because these subqueries are invariant under \( k \)-bisimulation. Subtrees with height \( h > k \) cannot be answered by the structural index. The maximum possible subtrees are preferred to achieve the best acceleration.

In the third step, these subtrees are substituted by subqueries which perform the same semijoins or antijoins on the reduced graph. The technique is similar to the technique we use for single semijoins and antijoins. For each semijoin or antijoin that we need to substitute, we select the foreign key constraint that matches with the original join condition. Then we replace the original semijoin or antijoin by a semijoin or antijoin with a relation in the structural index schema. Again, we replace primary key attributes with from\_id and replace foreign key attributes with to\_id, but now for both the left hand and right hand side of the semijoin or antijoin operation, except for the attributes from the relation that is the root of the join subtree. Those attributes are replaced by block\_id. For an example, consider the following query consisting of semijoins and antijoins:

\[
(r \bowtie (t \bowtie (w \bowtie x)) \bowtie ((s \bowtie u) \bowtie v))
\]

Figure 3.4a shows the join tree for this query. Figure 3.4b identifies the largest possible subtrees of height at most 1 in red (dotted line) and blue (dashed line). Using the following substitution, it can be used on a structural index for \( k = 1 \):

\[
\begin{align*}
  r \bowtie (t \bowtie (\sigma_\varphi(w))) & \bowtie (\sigma_\zeta(s)) \\
  \text{where } \varphi & = \exists x_i \in \text{index.x.w\_fkey}(x_i, \text{to\_id} = w.\text{block\_id}) \\
  \text{and } \zeta & = \exists u_i \in \text{index.u.s\_fkey}(u_i, \text{to\_id} = s.\text{block\_id}) \\
  & \wedge \neg\exists v_j \in \text{index.v.s\_fkey}(v_j, \text{to\_id} = s.\text{block\_id})
\end{align*}
\]

Figure 3.4c identifies the maximum subtrees of height at most 2. The following substitution can be applied for \( k = 2 \):

\[
\begin{align*}
  \sigma_\zeta(r) & \bowtie (\sigma_\varphi(t)) \\
  \text{where } \varphi & = \neg\exists w t_i \in \text{index.w.t\_fkey}(w t_i, \text{to\_id} = t.\text{block\_id}) \\
  & \wedge \exists x w j \in \text{index.x.w\_fkey}(x w j, \text{to\_id} = w t_i, \text{to\_id}) \\
  \text{and } \zeta & = \exists s r_k \in \text{index.s.r\_fkey}(s r_k, \text{to\_id} = r.\text{block\_id}) \\
  & \wedge \exists u s_l \in \text{index.u.s\_fkey}(u s_l, \text{to\_id} = s r_k, \text{from\_id}) \\
  & \wedge \neg\exists v s_m \in \text{index.v.s\_fkey}(v s_m, \text{to\_id} = s r_k, \text{from\_id})
\end{align*}
\]

In Figure 3.4d we see that the previous two subtrees are merged and that the maximum subtree of height at most 3 is equal to the tree itself. The following substitution can be used for \( k \geq 3 \):
3.5. QUERY ACCELERATION

Figure 3.4: Join subtree substitution

\[
\sigma_\varsigma(r) = \exists sr_i \in \text{index.r.fkey}(sr_i.to_id = r.block_id) \land \exists us_j \in \text{index.s.fkey}(us_j.to_id = sr_i.from_id) \\
\land \neg \exists vs_k \in \text{index.s.fkey}(vs_k.to_id = sr_i.from_id)) \land \exists tr_l \in \text{index.t.r.fkey}(tr_l.to_id = r.block_id) \\
\land \neg \exists wt_m \in \text{index.w.t.fkey}(wt_m.to_id = tr_l.from_id) \\
\land \exists xw_n \in \text{index.x.w.fkey}(xw_n.to_id = wt_m.to_id)))
\]

3.5.2 Equijoin acceleration

Equijoin operations can be simplified by limiting the joined relations to only participating tuples. A similar approach has been applied in distributed databases to speed up join processing by Stocker et al. in [18] and Sattler in [15]. Let \( r \) and \( s \) be relations that we want to join. Then we can substitute \( r \) by \((r \bowtie s)\) and substitute \( s \) by \((r \bowtie s)\) yielding the following equation

\[
r \bowtie s = (r \bowtie s) \bowtie (r \bowtie s).
\]

However, in case of a join on a one-to-many relationship with total participation, there is a foreign key with referencing attributes that are defined to be NOT NULL. Therefore, we can expect that for at least one of the relations, all tuples participate in the join. In that case, we can only reduce one relation in the join giving the following substitution:

\[
r \bowtie s = (r \bowtie s) \bowtie s.
\]

In case of multiple joins, we may be able to reduce the size of all participating relations. Consider the following joins on relations \( r, s \) and \( t \):

\[
r \bowtie_{r.A=s.B} s \bowtie_{s.B=t.C} t
\]

If \( s \) has total participation in each single join, but only a small subset of \( s \) participates in both joins, it can be interesting to compute the semijoins first, giving the following substitution:
Intuitively, joins become less expensive if one or more joined relations become smaller. Equijoin reduction using semijoins may accelerate join processing. Because we can also apply methods for semijoin substitution described in Subsection 3.5.1, structural indices may also be used for accelerating Equijoins.

3.5.3 Value-based selection conditions

It is very common for applications to combine semijoins or antijoins with value-based selection conditions. Value-based selections can be expressed using the $\sigma_\Theta$ operator. Consider the following selection on a semijoin:

$$\sigma_\Theta(r \bowtie s)$$

In this query, $\Theta$ serves as a selection predicate on $r \bowtie s$. When $\Theta$ contains only attributes from $r$, we may “push down” the selection to $r$. Then we can apply our semijoin substitution technique to accelerate the semijoin. We then filter $r$ on $\Theta$ in a sequential scan, and at the same time use our structural index to select tuples that have a partition block_id that participates in the join:

$$\sigma_\Theta(r) \bowtie s$$

However, when $\Theta$ contains attributes on $s$, we cannot use our structural index. We have to process the semijoin to determine which tuples satisfy the selection condition in $s$ participate in the join:

$$r \bowtie \sigma_\Theta(s)$$

A possible solution to still accelerate these queries is given in Section 3.6

3.6 Hybrid indexing

This section we present hybrid indexing. It combines the power of traditional value-based indexing with structural indexing. Section 3.3 described that the graph representation of the original database only captures structural information. Relation names, primary key values and foreign key values are part of the graph, but other attributes are ignored. As described in Subsection 3.5.3, some queries that include value-based selection predicates cannot be accelerated using the structural index. Consider the following query:

$$r \bowtie_{r.a=s.f} \sigma_{a=32}(s)$$

The structural index can tell which tuples in $r$ participate in a semijoin with $s$, but it does not distinguish between different values for attribute $a$. When a particular selection predicate is frequently used, we might want to include some attribute information in the structural index. A solution would be to reconsider our choice for the node labels. Instead of using only the relation name, we can combine the relation name with some attribute values. This ensures that the initial node partitioning under $k$-bisimulation is not only based on the relation name, but also based on one or more attributes. We can store the additional attributes in the ‘node’ type relations of the reduced graph to enable query acceleration. The example query can then be accelerated as follows:

$$r \bowtie_{r.a=s.f} \sigma_{a=32}(s) := \sigma_\varphi(r)$$

where $\varphi = \exists t_i \in \text{index.s.fkey}(t_i.to_id = r.block_id) \\
\land \exists u_j \in \text{index.s}(u_j.id = t_i.to_id \land u_j.a = 32)$
The performance of the structural index decreases when too many partition blocks are generated. Therefore, we must ensure that the attributes included in the index do not contain many distinct values. When the number of distinct values is more than 10, we can apply a function on the attribute values to reduce the number of distinct values. This solution is similar to features like list partitioning or range partitioning.

3.7 Workload-driven structural indexing

Our approach for the graph representation explained in Section 3.1 results in a $k$-bisimulation partitioning for the full database. This solution is useful for unknown queries when we can expect that a majority of the relations are accessed by semijoin and antijoin queries and the database processes mainly reads.

Not all queries can benefit from structural indices. Some relations are accessed more often than others. Moreover, the creation and maintenance of structural indices can require a significant amount of computation power. For most applications with database support, queries are known upfront. Therefore, it might be interesting to see if we can create smaller structural indices that only improve a specific set of queries. Then, our research question is as follows: “Given a set of queries, can we build a structural index specific for this workload?” This section presents three possible solutions.

Queries from end-user applications are often fixed or parameterized with user input. With respect to query knowledge, we can distinguish following cases:

1. Queries are known upfront
   (a) Set of concrete queries
   (b) Set of queries with parameter values based on user input

2. Queries are unknown

   It can be inefficient and unnecessary to create indices for the whole database if it is known upfront that only some parts will be used. Instead, we may create a structural index for some parts of the database. This section presents three alternative workload-specific approaches for structural indexing.

3.7.1 $k$-Bisimulation structural indexing on selected relations

We may select a particular set of relations and build a structural index for those relations. This alternative approach requires a small change in graph representation: we only generate nodes for tuples in selected relations and then generate edges between those nodes based on foreign key relationships. We can select those relations that are known to be accessed by queries that can benefit from structural indexing. This prevents the unnecessary computation and storage of bisimulation partitionings for relations where it is not beneficial.

This solution is suitable when it is known upfront which relations are expected to be accessed by semijoin and antijoin queries, but when exact query programs are unknown. For example, when it is known which relations occur in the query programs, but the query heads are unknown.

3.7.2 Relation partitioning on query result

Another approach would be to delay the generation of a structural index until it will be used. Based on the resultset of a tuple selecting join query, we can tag the resulting tuples in the original relation. When the same query is evaluated later on, we can directly return the tagged tuples without performing the join. This solution is a form of auto-tuning similar to the ideas described by Idreos in [4].

In case of parameterized queries, this solution may be less effective when every query is slightly different. This approach is useful when parameterized queries with the same parameters occur more often, when the initial index generation costs should be avoided and when no memory is available to cache query results.

3.7.3 DAG-based structural indexing

The former solutions use guarded $k$-bisimulation to propagate structural information within distance $k$ from the source relation. A guarded $k$-bisimulation iteration can be seen as a process that refines the partitioning for each relation in the schema based on the partitioning of other relations referenced by
foreign keys. The drawback of this approach is that a computed partitioning is only based on previous iterations, while a more recent partitioning for a particular relation may already have been created during the current iteration.

In the third alternative, we explicitly define the propagation of structural information through the schema by using a directed acyclic graph (DAG). Nodes represent relations and the edges specify the partitioning order. We use node labels to specify the relations and edge labels to specify the join conditions, which means that we are no longer restricted to key-foreign key joins. When a directed edge is exists from node \( n_a \) to node \( n_b \), this means that the relation of \( n_a \) is partitioned before the relation of \( n_b \) is partitioned. Because DAGs are acyclic, we can apply topological sorting to obtain a possible permutation \( p \) that defines the partitioning order. When a partitioning has been created for relation \( p_i \), we can directly partition relation \( p_{i+1} \) based on the partitioning \( p_i \) without waiting for the next iteration. The concept of ‘iterations’ is no longer required. Some relations may occur more often in \( p \). In that case, an existing partitioning is further refined.

The advantage of this approach can be described as follows: if we partition \( n \) relations one time, the structural information can propagate from the first relation to the last relation, with distance up to \( n \). Guarded \( k \)-bisimulation partitioning requires \( n \) guarded \( k \)-bisimulation iterations to propagate structural information with distance \( n \). This may require \( n \) times more computation power. The drawback is that the structural information propagates only in the chosen direction, which means that the resulting structural index is also less powerful than it is under guarded \( n \)-bisimulation, but still more powerful than only one guarded \( k \)-bisimulation iteration.

DAG based directional bisimulation partitioning is suitable when the exact query programs are known upfront. This solution requires expert knowledge about the queries and schema to determine beneficial structural index specifications.

We will now describe how a partitioning can be computed for a given DAG. Consider a database \( D \) and a DAG \( G = (V, E) \), where \( V \) denotes the set of labeled nodes and \( E \) denotes the set of labeled edges. The following algorithm creates a partitioning for \( D \) that follows the propagation structure defined in \( G \), using relation names as node labels and join conditions as edge labels.

**Algorithm** DAG-CREATE-PARTITIONING\((D,G=(V,E))\)

1: \( p = \text{TOPOLOGICAL-SORT}(G) \)
2: remove vertices from \( p \) that have no incoming edges in \( E \)
3: for \( i = 1 \) to \(|p| \) do
4: vertex \( v = p[i] \)
5: \( r = \text{relation in} \) \( D \) that satisfies \((r, \text{relation}_\text{name} == v.\text{label}) \)
6: for each tuple \( t \) in \( r \) do
7: if \((r \text{ has an existing partitioning}) \) then
8: \( \text{signature}[t] = \{\text{partition}, \text{block}, \text{id}[t]\} \)
9: else
10: \( \text{signature}[t] = \{\} \)
11: for each edge \( e \in E \) from vertex \( a \) to vertex \( b \) do
12: if \((v = b) \) then
13: \( s = \text{relation in} \) \( D \) that satisfies \((s, \text{relation}_\text{name} == a.\text{label}) \)
14: \( S_t = \text{list of tuples in} \) \( s \) for which the join condition in \( e.\text{label} \) matches tuple \( t \)
15: append \( \text{signature}[t] \) with a set of distinct partition block identifiers in \( S_t \)
16: create a partitioning based on the signatures in \( r \)
17: store the partitioning in \( r \)
18: mark \( r \) as partitioned
19: return \( D \)

**TOPOLOGICAL-SORT\((G)\)** is a function that computes a topological sort for a given graph \( G \). The output consists of a list of vertices that satisfies the topological ordering. This means that for every edge \( u \) to \( v \), \( u \) occurs before \( v \) in the topological ordering.

Line 7 to 14 generate a signature for a tuple \( t \) based on the partition block identifiers of connected tuples. The join predicates in the edge labels define which tuples are connected to \( t \). Any existing partition block identifier is part of the signature as well. This ensures that only refinements of the partitioning are possible. Line 15 creates a partitioning based on all signatures. Tuples with the same signature are assigned to the same partition block.

A reduced graph can be created using the methods described in Section 3.3. Then we can use the
methods for query acceleration described in Section 3.5.1 for joins where the join condition occurs as an 
edge label in the DAG.
Chapter 4

Experimental study

In this chapter, we present the results of an empirical analysis of the use of structural indexing following the approach for structural index creation and join acceleration presented in Chapter 3. In the following experiments, we measure how much query runtime speedup we can achieve if there exists a structural index for our database. We test both synthetic datasets and a real world dataset.

We will evaluate FACQ queries and guarded fragment queries. All queries will be strictly acyclic, which means that they are both acyclic and freely acyclic. We will focus at tuple selecting queries. A “tuple selecting” query has the restriction that the query result consists of a subset of the tuples from one relation. These queries can include semijoins or antijoins. We will also have a look at tuple generating queries which combine attributes from multiple multiple relations to create new tuples. These queries contain equijoins as well.

All experiments are conducted on a notebook with an Intel Core i5 520M 2.4 GHz processor, 8GB main memory and a Samsung 840 EVO 250GB flash drive processor. PostgreSQL 9.3 will be used as our database management system in a virtualized Ubuntu 14.04 LTS 64-bit environment with 5GB main memory and host I/O caching disabled.

Before each measurement, we stop the PostgreSQL server, flush the operating system caches and start the PostgreSQL server. This ensures that each query is executed with empty caches and buffers.

4.1 Pruning power evaluation for structural indexing

In the first experiment we measure how several dataset properties influence the potential acceleration factors for single semijoins, antijoins and equijoins. First of all, the difference in the size of two joined relations may influence the acceleration factor. Second, the amount of participating tuples may influence the acceleration factor. The influence of these properties might be different for semijoins, antijoins and equijoins.

The creation of a structural index is not part of this experiment. We also neglect the cost of accessing a reduced graph to determine which partition blocks participate in the join. Instead, we assume that we know for each tuple whether it participates in a particular join by tagging those tuples. We do this by storing the participation knowledge in a participation attribute during the dataset generation. When a tuple participates in a join, we set the participation attribute to 1. Otherwise, we set the participation attribute to 0. For every tuple we can then determine if it participates in the join by comparing the value of the participation attribute with value 1. A positive outcome means that the tuple participates in the join.

4.1.1 Data

We introduce two definitions to describe selectivity and size properties of our dataset:

\[
\text{selectivity factor } SF(r, s) = \frac{|r \bowtie s|}{|r|} \\
\text{size ratio } SR(r, s) = \frac{|r|}{|s|}
\]
The selectivity factor describes how selective a particular query is. In case of join operations, the join participation factor determines the selectivity of the join. For semijoins and equijoins, a high join participation factor results in a high selectivity factor: most records are part of the query result, so the query is not selective. For antijoins, a high join participation factor means a low selectivity factor: most tuples would be joined, so few tuples are part of the antijoin query result.

We construct relations \( r(a, b, c) \) and \( s(d, e, f) \). Suppose there is a foreign key from \( s(f) \) referencing \( r(a) \). Let \(|s| = 5 \cdot 10^6\) and \(|r| = \text{size}\_\text{ratio} \cdot |s|\). Further, we construct B-tree indices on primary keys and the foreign key. All attributes are configured to be \text{NOT NULL}. As a result, relation \( s \) has total participation in the join.

Synthetic datasets are generated for various selectivity factors and size ratios. Relation \( s \) contains 5 million tuples and the amount of tuples in \( r \) is determined by the size ratio factor. Participating and non-participating tuples are uniformly distributed over all disk blocks using Mersenne Twister 19937, a well-known pseudorandom number generator from Matsumoto et al. [10]. We use the following algorithm to create the dataset: while \(|s| < 5 \cdot 10^6\), generate a tuple \( \tau \) and, in case of participation, \( \frac{1}{SF \cdot SR} \) referencing tuples \( \sigma \). Using a bernoulli trial, we determine if \( \tau \) participates in the join with probability \( SF \). In case of a positive outcome, the referencing tuples \( \sigma \) are generated.

### 4.1.2 Running time evaluation for single joins

We measure the semijoin, antijoin and join acceleration by loading the generated dataset into PostgreSQL 9.3, \texttt{ANALYZE} the relations, and then use \texttt{EXPLAIN (ANALYZE TRUE, TIMING FALSE)} to measure the query performance. This SQL expression measures the total runtime of the query and shows the query plan. The transfer costs to the client and display costs are not measured. We do not use \texttt{EXPLAIN ANALYZE} because it measures the running time of individual nodes in the query plan, which may significantly increase the total running time due to timing overhead.

#### Semijoins and antijoins

Listing 4.1 shows a semijoin query and Listing 4.2 shows an accelerated version of the same query following the approach described in Subsection 3.5.1. Similarly, Listing 4.3 shows an antijoin query and Listing 4.2 shows the accelerated version of that query.

```sql
SELECT a, b
FROM r
WHERE EXISTS (SELECT 1 FROM s WHERE s.f = r.a)
Listing 4.1: Original semijoin

SELECT a, b
FROM r
WHERE r.b = 1
Listing 4.2: Accelerated semijoin

SELECT a, b
FROM r
WHERE NOT EXISTS (SELECT 1 FROM s WHERE s.f = r.a)
Listing 4.3: Original antijoin

SELECT a, b
FROM r
WHERE r.b = 0
Listing 4.4: Accelerated antijoin
```

Figure 4.1 and Figure 4.2 show the speedup in a heatmap of semijoin and antijoin queries for selectivity factors and size ratios from 10% up to 100%. We can observe that the speedup is maximal for both semijoin and antijoin queries when the size ratio is very small. The acceleration is around 20 times for size ratio 0.1. For semijoins, a slightly better speedup is found for low selectivity factors, while a high selectivity factor is slightly better for antijoins.
CHAPTER 4. EXPERIMENTAL STUDY

Equijoins

Listing 4.5 shows an equijoin query and Listing 4.6 shows an accelerated version of the same query following the approach described in Subsection 3.5.2. Figure 4.3 shows the speedup for single equijoins. The query planner chooses a query plan that has either a merge join or hash join. For most selectivity factors and size ratios, the query planner chooses a hash join for both the original and accelerated queries. However, for size ratios 0.3, 0.6, 0.9 and 1.0, the query planner prefers merge joins for the original query but hash joins for the accelerated versions. For all selectivity factors and size ratios, merge joins outperform hash joins. As a result, some cells show a significant slowdown. For selectivity 0.9 with size ratio 0.7, this is exactly the other way around.
4.1. PRUNING POWER EVALUATION FOR STRUCTURAL INDEXING

Listing 4.5: Original join

```sql
SELECT a, b, c, d, e, f
FROM r
INNER JOIN s
ON s.f = r.a
```

Listing 4.6: Accelerated join

```sql
SELECT a, b, c, d, e, f
FROM r
INNER JOIN s
ON s.f = r.a
WHERE r.b = 1
```

Figure 4.3: Single join speedup

Figure 4.4 shows the speedup when the query planner is forced to use hash joins. In case of hash joins with low selectivity, the substitution methods gives slightly better performance.

Figure 4.5 shows the speedup when the query planner is forced to use merge joins. The accelerated query does not make a significant difference and shows a slight slowdown in most cases. The reason is that it requires an additional sort operation. The original query can uses index scans using B-trees on the join attributes for both relations to obtain a sorted input for the merge join. In the accelerated version, the B-tree is not being used due to the selection condition. Figure 4.6 shows the differences between both query plans.

4.1.3 Discussion

We can observe that the size ratio has the biggest influence on the acceleration factor for semijoins and antijoins. The selectivity factor has only a very small influence. The best acceleration is found when we select a small amount tuples from a small relation that is semijoined or antijoined with a much larger relation.

For equijoins, we observed that a hash join queryplan can be slightly accelerated by reducing the join to participating records. However, when B-tree indexes are built on primary keys and foreign keys, merge join query plans are still faster than both original and accelerated hash join query plans and cannot benefit from the given acceleration methods. For the generated datasets, there is no equijoin acceleration possible when the best query plan without participation knowledge is compared with the the best query plan that uses the join participation knowledge.

Equijoins are hard to accelerate by tagging participating tuples for the following reasons. First of all, all joined relations must be accessed to process equijoins. Semijoins and antijoins can read just one relation and output all tagged tuples. Further, because all the participating records are uniformly distributed over all disk blocks that store the joined relations, it is likely that most of these disk blocks have to be accessed anyway. Limiting the join to participating records does not save I/O costs.
4.2 Implementation for $k$-Bisimulation structural indexing

Several tools have been developed to allow the creation of a structural index for a PostgreSQL database. Our toolset is based on the existing $\texttt{kbisim\_opt}$ implementation for the external memory $k$-bisimulation partitioning algorithm that Luo et al. developed as part of their work in [8]. Our toolset supports multi-attribute primary keys and multi-attribute foreign keys that have distinct names for referencing and referenced attributes.

The first tool called $\texttt{postgres2edges}$ was developed in C++ to extract the data definition language (DDL) from an arbitrary PostgreSQL database based on the ‘information_schema’ catalog. It generates
4.3. K-BISIMULATION STRUCTURAL INDEXING FOR TPC-H

In this experiment, we test our methods for semijoin and antijoin substitution on TPC-H, a synthetic dataset that is often used for relational database management system benchmarks. It is a common practice to build indices on foreign keys. The goal of this experiment is to measure whether a structural index can give faster query performance than B-tree indices on foreign keys.

Every measurement was executed three times and the mean values are used for plotting. In PostgreSQL, the settings for shared_buffers and work_mem were increased to 1 GB to improve the sort and join performance. The Genetic Query Optimizer (geqo) was disabled and the Query Planner Statistics (default_statistics_target) was increased to 10000 to reduce the chance of bad query plans.

4.3.1 Data

The TPC-H benchmark version 2.17.1 was used to test our approach. This benchmark is an industry standard for benchmarking relational database management systems. It contains a tool called DBgen to...
CHAPTER 4. EXPERIMENTAL STUDY

Figure 4.7: Processing workflow showing the application of all tools

create synthetic data. The relational model represents a business “trade” database that stores parts, suppliers, customers and orders. The size of the dataset is determined by a scale factor. We used scale factor $SF = 1$ for our evaluation. This generates eight tables with 8,402,291 records in total. TPC-H also contains a tool `qgen` that generates queries from 22 predefined formats, including several join-intensive queries. These queries may retrieve attributes from multiple relations, contain selection conditions on attributes from relations other than retrieved attributes and have complex aggregations. As a result, these queries cannot directly be answered on structural indices and may benefit from traditional indexing techniques to improve query performance. Therefore, we will restrict ourselves to semijoins and antijoins and we write our own queries based on the reduced graphs.

Table 4.1 shows the size of each relation and Figure 4.8 shows the schema of foreign key edges. The dataset contains no trees or advanced data structures.

<table>
<thead>
<tr>
<th>Relation</th>
<th>tuples</th>
<th>table size</th>
<th>index size</th>
<th>total size</th>
</tr>
</thead>
<tbody>
<tr>
<td>customer</td>
<td>150000</td>
<td>29MB</td>
<td>3MB</td>
<td>32MB</td>
</tr>
<tr>
<td>lineitem</td>
<td>6001215</td>
<td>949MB</td>
<td>309MB</td>
<td>1258MB</td>
</tr>
<tr>
<td>nation</td>
<td>25</td>
<td>8kB</td>
<td>16KB</td>
<td>24KB</td>
</tr>
<tr>
<td>orders</td>
<td>1500000</td>
<td>215MB</td>
<td>32MB</td>
<td>247MB</td>
</tr>
<tr>
<td>part</td>
<td>200000</td>
<td>32MB</td>
<td>4MB</td>
<td>36MB</td>
</tr>
<tr>
<td>partsupp</td>
<td>800000</td>
<td>143MB</td>
<td>24MB</td>
<td>167MB</td>
</tr>
<tr>
<td>region</td>
<td>5</td>
<td>8kB</td>
<td>16KB</td>
<td>24KB</td>
</tr>
<tr>
<td>supplier</td>
<td>10000</td>
<td>1840kB</td>
<td>240KB</td>
<td>2080KB</td>
</tr>
<tr>
<td>$\sum$</td>
<td>8402291</td>
<td>1370MB</td>
<td>372MB</td>
<td>1742MB</td>
</tr>
</tbody>
</table>
4.3.2 Experimental design

The DBgen tool was compiled and used with scale factor 1 to create the smallest possible dataset. This dataset was loaded into PostgreSQL 9.3 using the tutorial at http://web.archive.org/web/20140805041546/http://www.fuzzy.cz/en/articles/dss-tpc-h-benchmark-with-postgresql/. The schema of the database is included in [8] for reference purposes.

A structural index for the PostgreSQL database was created using the toolset that we will describe in Section 4.2. We processed \( k \)-bisimulation iterations until no new partition blocks were generated. Further, we visualized the reduced graphs for the first four iterations. Then we verified that the structure of the graph represents the structure of the original database. Finally, we answer several queries using the structural index and discuss the acceleration factor.

4.3.3 Reduced graph analysis

Figure 4.9 shows for each relation the number of partition blocks that are generated under \( k \)-bisimulation for \( 1 \leq k \leq 12 \). For \( k \geq 12 \), the total number of partition blocks is 8,611,255 which is exactly the number of tuples in the database. This means that none of the nodes are fully bisimilar. In this case, full bisimulation results in an index that has the same size as the original dataset. Localized bisimulation solves this problem.

In the rest of this subsection, we show visualizations of reduced graphs for the first four \( k \)-bisimulation iterations. For each graph, we discuss which information can be retrieved from the graph and what is hidden.

\( k \)-Bisimulation is a localized approximation of normal bisimulation where topological features of nodes are considered within a neighborhood of radius \( k \), where \( k \) also denotes the number of iterations [8]. This means that for \( k \geq 0 \) iterations, the existence of features can only be confirmed up to distance \( d \leq k \) from a particular partition based on the reduced graph. On the contrary, we can reject the existence of features for arbitrary distance \( d > 0 \) by interpreting the reduced graph. The existence of features for \( d > k \) can only be rejected and is indeterminate otherwise.

Figure 4.10 shows the partitioning of the TPC-H dataset under \( 0 \)-bisimulation. This graph does not give more information than the schema. However, there are still queries that can be answered from the structure. Because all foreign keys are NOT NULL, we can state that all lineitems are connected to regions via orders, customers and nations. Moreover, these tuples are also connected via partsupps, suppliers and nations.

Figure 4.11 shows the partitioning of the TPC-H dataset under \( 1 \)-bisimulation. The graph resembles...
Figure 4.9: Number of partition blocks for TPC-H tuples after $k$-bisimulation partitioning

Figure 4.10: TPC-H $k$-bisimulation partitioning with $k = 0$
the graph of the schema but now there are two customer partition blocks and two partsupp partition blocks. Partition customer-1 contains the tuples in the customer relation for which no tuple exists in the order relation that point to these customer tuples. On the contrary, customer-2 contains the tuples in the customer relation for which one or more tuples do exist in the order relation that point to these customer tuples. Concretely this means that some customers have ordered something while other customers do not. Similarly, partition partsupp-1 contains the tuples that ‘have’ one or more lineitems and partsupp-2 contains the tuples that do not have a lineitem.

Based on the first bisimulation iteration, we cannot tell whether all parts in partition part-1 are connected to a lineitem via partsupp-1. It may be the case that one or more parts in part-1 are only referred from partsupp-2 and are therefore not connected to a lineitem. Similarly, we cannot tell whether all suppliers in supplier-1 are connected to a lineitem.

Figure 4.11: TPC-H $k$-bisimulation partitioning with $k = 1$

Figure 4.12 shows the partitioning of the TPC-H dataset under 2-bisimulation. Based on the partition part-2, we can now confirm that there are no parts that are not connected to a lineitem via some partsupps. Moreover, we can confirm that there are no suppliers that are not connected to a lineitem. This information was not available from the first iteration.

Figure 4.12: TPC-H $k$-bisimulation partitioning with $k = 2$

Figure 4.13 shows the partitioning of the TPC-H dataset under 3-bisimulation. There are now more
partsupp partition blocks than in the previous partitioning. For each partsupp partition, we will explain its properties. partsupp-1 contains the tuples that are not directly connected to a lineitem, but its suppliers and parts do have other partsupps that do have a direct connection to a lineitem. Tuples in partsupp-2 are directly connected to a lineitem. Further, all of its suppliers and parts only have partsupps that are directly connected to a lineitem. Partition partsupp-3 contains tuples that are directly connected to a lineitem. Similar to the former partsupp, all of its parts have only partsupps that are directly connected to a lineitem. On the contrary, it does have suppliers that have partsupps that are not directly connected to a lineitem. Partition partsupp-4 is exactly the other way around. All of its suppliers have only partsupps that are directly connected to a lineitem but it has parts that have other partsupps that are not directly connected to a lineitem. Partition partsupp-5 has both suppliers and parts that have other partsupps that are not directly connected to a lineitem.

Figure 4.13: TPC-H $k$-bisimulation partitioning with $k = 3$

Figure 4.14 shows the partitioning of the TPC-H dataset under $4$-bisimulation. Without discussing every partition, we can see that the original differences in partsupps from the first iteration are now propagated to lineitems as well. Further, we can see some zigzag effect between parts, suppliers and partsupps. There are now four part partition blocks instead of two. Both part-1 and part-2 are only directly connected to partsupps that have a direct connection with lineitems. The difference is that parts in part-1 are only supplied by suppliers that have only partsupps that are directly connected to lineitems while parts in part-2 are also supplied by suppliers that have partsupps that are not directly connected to lineitems.

### 4.3.4 Structural Index Size Analysis

Figure 4.15 shows the size of the structural index in addition to the original TPC-H database for $0 \leq k < 12$ iterations. The additional size that would be required for B-tree indices on foreign keys is added to the plot as well. The size was measured after the index was loaded into the PostgreSQL database and a full vacuum operation was executed. A full vacuum was required because updating all block_id values doubles the size of all relations due to MVCC support. The partitioning column uses 34 megabyte which is less than 2% of the original database. The reduced graph uses only 1 megabyte for the first seven iterations. For iterations 8, 9, 10 and 11 the sizes are respectively 66 MB, 1026 MB, 1679 MB and 1700 MB. We can observe that the structural index uses less space than B-tree indices on foreign keys for $k < 9$.

### 4.3.5 Query Performance Analysis

In this section, we analyze the performance improvement for several types of queries. We prepared 11 databases with a structural index for $1 \leq k \leq 11$. We compare structural indexing using $k$-bisimulation partitioning with a traditional indexing techniques: B-tree indices on primary keys and foreign keys. To improve the running time, we manually substitute query parts that can be answered by the structural
Figure 4.14: TPC-H $k$-bisimulation partitioning with $k = 4$
CHAPTER 4. EXPERIMENTAL STUDY

Figure 4.15: Size of structural index in addition to the original TPC-H database

index using the subtree substitution algorithm described in Section 3.5. Then we compared the running time of the accelerated query with the original query executed on the original database. The original SQL queries and accelerated queries can be found in Appendix C.

Freely Acyclic Conjunctive Queries

Table 4.2 lists the Freely acyclic conjunctive queries (FACQ) queries that have been tested on the TPC-H dataset. FACQ queries are explained in Section 2.4.

Table 4.2: FACQ queries for TPC-H dataset

<table>
<thead>
<tr>
<th>Query</th>
<th>join height</th>
<th>selectivity</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FACQ 1A</td>
<td>1</td>
<td>0.66</td>
<td>Select customer names having an order</td>
</tr>
<tr>
<td>FACQ 1B</td>
<td>1</td>
<td>0.99</td>
<td>Select partsupp supplycosts having a lineitem</td>
</tr>
<tr>
<td>FACQ 2A</td>
<td>2</td>
<td>1</td>
<td>Select supplier names having a lineitem</td>
</tr>
<tr>
<td>FACQ 2B</td>
<td>2</td>
<td>1</td>
<td>Select part names having a lineitem</td>
</tr>
<tr>
<td>FACQ 4A</td>
<td>4</td>
<td>1</td>
<td>Select region having nation, customer, orders and lineitem</td>
</tr>
</tbody>
</table>

Figure 4.16 shows the performance improvement for each query. Note that all queries can be expressed using semijoins. Inner joins are not included in this experiment. Queries 1A and 1B consist of only one semijoin. Their maximum speedup is 5 times for query 1A and 12 times for query 1B. We can see in the graph that the speedup significantly decreases for $k \geq 8$ and $k \geq 9$.

Queries 2A and 2B perform two nested semijoins. The first iteration already provides acceleration, however, their maximum performance is reached at the second bisimulation iteration where 2A reaches 26 times and 2B reaches 16 times. The reason for this is that for $k = 1$, only one semijoin is accelerated. For $k \geq 2$, both semijoins are accelerated. The speedup decreases for $k \geq 8$ due to the larger reduced graph size. The performance of queries 2A and 2B drop for $k \geq 9$.

Query 4A contains four nested semijoins with join height 4. For this reason, the performance is maximal for $k \geq 3$ when most of the semijoins can be accelerated. The maximum speedup is 33 times.

Guarded Fragment Queries

Table 4.3 lists the guarded fragment queries that are tested on the TPC-H dataset. Guarded fragment queries are explained in Section 2.4.
4.3. K-BISIMULATION STRUCTURAL INDEXING FOR TPC-H

Figure 4.16: Running time acceleration for FACQ queries using $k$-bisimulation partitioning on TPC-H

Table 4.3: Guarded fragment queries for TPC-H dataset

<table>
<thead>
<tr>
<th>Query</th>
<th>join height</th>
<th>selectivity</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GF 1A</td>
<td>1</td>
<td>0.0006</td>
<td>Select partsupp comment not having a lineitem</td>
</tr>
<tr>
<td>GF 1B</td>
<td>1</td>
<td>0.3334</td>
<td>Select customers not having an order</td>
</tr>
<tr>
<td>GF 2A</td>
<td>2</td>
<td>0.0023</td>
<td>Select parts having partsupp without lineitem</td>
</tr>
<tr>
<td>GF 4A</td>
<td>4</td>
<td>0.1285</td>
<td>Select all suppliers that sell parts that are offered but not sold by other suppliers</td>
</tr>
</tbody>
</table>
Figure 4.17 shows the performance speedup for each query. Queries 1A and 1B have only one antijoin. Therefore, the maximum performance is achieved at the first iteration. The acceleration is around 9 times for 1A and around 4 times for 1B. Queries 2A and 4A combine a semijoins with antijoins. Their maximum speedup of 14 times and 173 times is reached at the second and fourth iteration. For $k \geq 8$, the speedup starts to drop for all four queries.

![Figure 4.17: Running time acceleration for guarded fragment queries using $k$-bisimulation partitioning on TPC-H](image)

**Value-based Selective Queries**

Queries may also combine selection conditions with semijoins and antijoins. Query VS 1A was designed to test the performance when only a few tuples are retrieved. Query VS 2A tests the performance when thousands of tuples are retrieved. Table 4.4 lists the queries that are used to measure the relation between the number of retrieved tuples and the runtime performance. Figure 4.18 shows that the performance using the structural index is comparable to using the btrees on primary keys and foreign keys when less than 20 tuples are retrieved. Figure 4.19 shows that the structural indices for $k \geq 8$ give a higher and more stable performance with an increasing amount of retrieved tuples, while the performance using traditional slightly varies with an increasing amount of retrieved tuples. Indices for $k \geq 10$ are slower when only few tuples are retrieved.

**Table 4.4: Value-based selective queries for TPC-H dataset**

<table>
<thead>
<tr>
<th>Query</th>
<th>join height</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VS 1A</td>
<td>1</td>
<td>Select customers having order where account balance below variable amount</td>
</tr>
<tr>
<td>VS 2A</td>
<td>2</td>
<td>Select part names having lineitem and part below variable size</td>
</tr>
</tbody>
</table>

**4.3.6 Discussion**

In this section, we experimented with several types of queries and applied our acceleration methods on a relational database. For the TPC-H dataset, we can conclude that structural indexing can give around 10 times better running time performance for the tested FACQ and guarded fragment queries.
4.3. K-BISIMULATION STRUCTURAL INDEXING FOR TPC-H

Figure 4.18: Running time for value-based selective query VS1A

Figure 4.19: Running time for value-based selective query VS2A
with $1 < k < 8$. Higher acceleration factors up to 190 times are possible for queries that have multiple joins. At the same time, these structural indices use significantly less disk space than B-tree indices on foreign keys. For $k < 8$, the size of the structural index is less than 2% of the original database. No single FACQ or guarded fragment query showed a slowdown when the structural index for $2 \leq k \leq 8$ was used.

Structural indices are not designed to accelerate value-based selective queries. However, the experiments for these queries showed that small accelerations are still possible, but not guaranteed for under high $k$-bisimulation indexing when less than 5000 tuples are returned. The experiments for these type of queries showed that a larger resultset influences the running time for the original query plans, but queries that use the structural index with $k \leq 8$ show a stable running time, regardless of the amount of tuples in the resultset.

We can conclude that structural indexing is beneficial for FACQ and guarded fragment queries on the TPC-H dataset. It outperforms traditional indexing techniques. At the same time, our approach uses less disk space usage than current solutions.

### 4.4 Hybrid indexing on TPC-H

In this section, we demonstrate how structural indexing can be combined with traditional value-based indexing techniques. The goal is to accelerate queries with value-based selection conditions nested in a semijoin or antijoin. To demonstrate this technique, we created a hybrid structural index for the TPC-H dataset. The attribute `o.orderstatus` from the `orders` relation was included in the index. This allows to accelerate semijoin or antijoin queries that have a selection on the order status attribute. There are three possible values for the orderstatus: Open (O), Pending (P) or Finished (F).

Every measurement was executed three times and the mean values are used for plotting. In PostgreSQL, the settings for `shared_buffers` and `work_mem` were increased to 1 GB to improve the sort and join performance. The Genetic Query Optimizer (geqo) was disabled and the Query Planner Statistics (default_statistics_target) was increased to 10000 to reduce the chance of bad query plans.

#### 4.4.1 Implementation

The implementation for $k$-bisimulation described in Section 4.2 was used for hybrid structural indexing with a small adaptation. Instead of labelling the nodes in the graph representation with the relation name, we also included the order status for nodes that represent the order relation. So instead of `orders`, we used `orders-O`, `orders-P` and `orders-F` as node labels in the graph representation. As a result, the `textttedges2postgres` executable also generates three separate order relations to store the order nodes from the resulting reduced graph. Using a small query, we manually merged the three order relations into one node relation with an additional order status attribute.

#### 4.4.2 Reduced graph analysis

![Figure 4.20: Hybrid $k$-bisimulation partitioning on TPC-H with orderstatus for $k = 1$](image-url)
Figure 4.20 visualizes the partitioning for the structural index under $1$-bisimulation. The partitioning is different compared to the regular partitioning for TPC-H in Figure 4.11. We now have three different order partition blocks: one partition block for each order status. Moreover, there are now eight customer partition blocks. Customers can now have three types of orders, which results in $2^3 = 8$ possible combinations.

We generated a hybrid structural index for $k = 5$. The number of nodes is 501,799 compared to only 15 nodes for a regular structural index for TPC-H with $k = 5$.

4.4.3 Query performance analysis

Table 4.5 lists the queries that were performed on the TPC-H dataset. These queries contain nested selection conditions and therefore they cannot be evaluated on a regular structural index that does not contain attributes. We evaluated each query on the original database and on a hybrid structural index for $k = 3$. The SQL queries are added to Appendix C.4. Queries HYBRID 1A and HYBRID 1B show an acceleration factor of respectively 8.85 and 10.09 times. Query HYBRID 2A shows a very limited acceleration factor of 1642 times, but HYBRID 3A gives an acceleration only 23.59 times. The accelerations using hybrid structural indices are comparable or better than for FACQ and guarded fragment queries, however, the performance drops for $k \leq 4$. The reduced graph for a hybrid structural index contains more nodes and edges than a regular structural index, which slows down joins on the structural index.

<table>
<thead>
<tr>
<th>Query</th>
<th>height</th>
<th>selectivity</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HYBRID 1A</td>
<td>1</td>
<td>0.20</td>
<td>Select names of customers having a pending order</td>
</tr>
<tr>
<td>HYBRID 1B</td>
<td>1</td>
<td>0.66</td>
<td>Select names of customers having a completed order</td>
</tr>
<tr>
<td>HYBRID 2A</td>
<td>2</td>
<td>1</td>
<td>Select names of nations having a customer with pending order</td>
</tr>
<tr>
<td>HYBRID 3A</td>
<td>3</td>
<td>0.61</td>
<td>Select parts having a pending order through partsupp and lineitem</td>
</tr>
</tbody>
</table>

4.4.4 Discussion

In this section, we demonstrated how structural indexing can be combined with value-based indexing. We showed a hybrid structural index for the TPC-H dataset and accelerated some queries. The acceleration factor is much smaller than for regular $k$-bisimulation structural indexing, but it is still possible to create a faster index than traditional B-tree indices.

4.5 $k$-Bisimulation structural indexing for DBLP

In this experiment, we test our methods for semijoin and antijoin acceleration on a real world dataset: DBLP. The DBLP dataset is a bibliography that lists more than 3 million computer science publications. The implementation from Section 4.2 was also used for this experiment on the DBLP dataset.

Every measurement was executed three times and the mean values are used for plotting. In PostgreSQL, the settings for shared buffers and work mem were increased to 1 GB to improve the sort and join performance. The Genetic Query Optimizer (geqo) was disabled and the Query Planner Statistics (default_statistics_target) was increased to 10000 to reduce the chance of bad query plans.

4.5.1 Data

DBLP provides an XML file that contains the bibliography database [7]. It includes the following publication types: article, inproceedings, proceedings, book, incollection and PHD thesis. For each publication, it contains the type, name, year, list of authors and other information depending on the type. For each author, it captures the name and homepages. For this experiment, we restrict the publication types to articles, books, incollections and inproceedings. This means that not all publications will be used for the experiment, but we will include all authors. Table 4.6 shows the size of each relation and Figure 4.21 shows the schema of foreign key edges.
4.5.2 Experimental design

The XML file for the DBLP dataset was transformed to a relational database that can be loaded in PostgreSQL. PostgreSQL is an object-relational database and therefore it supports inheritance. This can be used to apply a generalization concept and let articles, books, incollections and inproceedings relations inherit some attributes from a publication relation. However, in PostgreSQL, it cannot be used in combination with foreign keys. Foreign keys are a must have for our structural indexing approach. As a result, we cannot use inheritance concepts that are available in PostgreSQL. We designed four separate entities for each publication type, an author entity, and four ‘authored’ many-to-many relations from each publication type to the author entity. The relational schema can be found in Appendix D.

A structural index for the PostgreSQL database was created using the toolset that was described in Section 4.2. We processed the first twelve \( k \)-bisimulation iterations. Further, we visualized the reduced graph for the first iteration. Finally, we answer several queries using the structural index and discuss the acceleration factor.

4.5.3 Reduced graph analysis

Figure 4.22 shows for each relation the number of partition blocks that are generated under \( k \)-bisimulation for \( 1 \leq k \leq 12 \). For \( k \geq 12 \), the total number of partition blocks is 8,973,686. From there on, it slowly increases to 9,321,512 partitions for \( k = 20 \). The total number of tuples is 13,154,516 which means that there are still some bisimilar tuples after twenty \( k \)-bisimulation iterations.

The total number of partition blocks for DBLP increases faster with respect to \( k \) compared to the TPC-H dataset. At \( k = 4 \), TPC-H has only 22 partition blocks while DBLP already has 1451 blocks. At \( k = 8 \), TPC-H shows super exponential partitioning size growth for \( 5 \leq k \leq 8 \), while DBLP shows super exponential partitioning size growth for \( 3 \leq k \leq 5 \). For this reason, we can expect that queries on the structural index start slowing down for lower \( k \) values than with TPC-H.

Figure 4.23 shows the partitioning of the DBLP dataset under \( 0 \)-bisimulation. The visualization gives the same information as the schema. We cannot answer the question whether all authors have a publication in our dataset, or whether all publications belong to an author in our dataset. This information is only available from \( k = 1 \).

Figure 4.24 shows the partitioning of DBLP under \( 1 \)-bisimulation. We can now confirm that most authors have a publication, but not all of them. Similarly, for each publication type, there are some
Figure 4.22: Number of partition blocks for DBLP tuples after $k$-bisimulation partitioning
publications that do not belong to an author. Our dataset contains four types of publications, so authors can write \(2^2 = 16\) combinations of publication types. There are 16 partition partition blocks for authors so we can conclude that every combination occurs in our dataset.

\[\text{article-1} \quad \text{size: 1318984} \quad \text{authored_article-1} \quad \text{size: 3548454}\]

\[\text{book-1} \quad \text{size: 11916} \quad \text{authored_book-1} \quad \text{size: 17857}\]

\[\text{incollection-1} \quad \text{size: 35114} \quad \text{authored_incollection-1} \quad \text{size: 59418}\]

\[\text{inproceedings-1} \quad \text{size: 1649100} \quad \text{authored_inproceedings-1} \quad \text{size: 4894795}\]

\[\text{author-1} \quad \text{size: 1618793}\]

Figure 4.23: DBLP \(k\)-bisimulation partitioning with \(k = 0\)

### 4.5.4 Structural index size analysis

Figure 4.25 shows the size of the structural index in addition to the original DBLP database for \(0 \leq k \leq 12\) iterations. The additional size that would be required for B-tree indices on foreign keys is added to the plot as well. The size was measured after the index was loaded into the PostgreSQL database and a full vacuum operation was executed. A full vacuum was required because updating all block id values doubles the size of all relations due to MVCC support. The partitioning column uses 16 megabyte which is less than 2% of the original database. The reduced graph uses only 1 megabyte for the first four iterations. For iterations 5, 6, 7 and 8 the sizes are respectively 56 MB, 145 MB, 566 MB and 1146 MB. We can observe that the structural index uses less space than B-tree indices on foreign keys for \(k < 7\).

### 4.5.5 Query performance analysis

In this section, we analyze the query acceleration for FACQ and guarded fragment queries. We constructed twelve databases with structural indices for \(1 \leq k \leq 12\). Similarly to our experiment with TPC-H, we compare the performance of structural indices with B-trees on foreign keys. Again, we designed some example queries and then manually rewrote these queries using our subtree substitution algorithm described in Section 3.5.1. All original and accelerated queries can be found in Appendix E.

**Freely Acyclic Conjunctive Queries**

<table>
<thead>
<tr>
<th>Query</th>
<th>join height</th>
<th>selectivity</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FACQ 1P</td>
<td>1</td>
<td>0.0078</td>
<td>Select names of authors who wrote books</td>
</tr>
<tr>
<td>FACQ 1Q</td>
<td>1</td>
<td>0.0016</td>
<td>Select names of authors who wrote books and incollections</td>
</tr>
<tr>
<td>FACQ 1R</td>
<td>1</td>
<td>0.0016</td>
<td>Select names of authors who wrote all publication types</td>
</tr>
<tr>
<td>FACQ 3P</td>
<td>3</td>
<td>0.1380</td>
<td>Select titles of articles that have an author who also wrote books</td>
</tr>
<tr>
<td>FACQ 3R</td>
<td>3</td>
<td>0.6897</td>
<td>Select titles of books that have an author who also wrote articles</td>
</tr>
<tr>
<td>FACQ 3S</td>
<td>3</td>
<td>0.6266</td>
<td>Select titles of books that have an author who also wrote articles and inproceedings</td>
</tr>
<tr>
<td>FACQ 5P</td>
<td>5</td>
<td>0.0905</td>
<td>Select names of authors who (co)authored an article that has an author who also wrote books</td>
</tr>
</tbody>
</table>

Table 4.7 lists the FACQ queries that are tested on our DBLP dataset. Queries FACQ 3P, 3R and 3S are listed as join height 3. However, these queries can be expressed in SQL with join tree height 2 on the original database. A FK-FK is used to write the query with join tree height 2: it joins a foreign key from
Figure 4.24: DBLP $k$-bisimulation partitioning with $k = 1$
an ‘authored’ relation with a foreign key from an ‘authored’ relation. When we restrict joins to PK-FK joins, this join height would not be possible. There are no edges for FK-FK joins in the structural index. As a result, these queries cannot be substituted by a query on the structural index with join height 2, but it is still possible to express the same queries on the original database with join height 3 and then accelerate one or more joins using the structural index. A similar situation exists for query FACQ 5P which can be expressed on the original database with join height 3 when two FK-FK joins are used.

Our goal is to compare the fastest query plan on the original database with the fastest query plan using the structural index. Therefore, we allow every possible join in the original queries and restrict ourselves to PK-FK joins on the structural index. Consequently, the accelerated query on the structural index is not necessarily faster than the original query, especially when the join trees are only partially substituted.

Figure 4.26 shows the acceleration factor for the FACQ queries that were tested on the DBLP dataset. We can observe that all queries with join height 1 are between two and eight times faster on the structural index for \( k \leq 4 \). For \( k > 4 \), the acceleration starts to drop for these queries. FACQ 1R no longer shows acceleration for \( k \geq 8 \).

For queries with join height 3 or 5, there is no significant acceleration for \( k = 1 \). The reason is that these queries access the relatively large ‘author’ relation which is omitted in the original query. Their maximum performance is found when their join height equals \( k \) and the query can be fully accelerated on the structural index.

Guarded Fragment Queries

Table 4.8: Guarded fragment queries for DBLP dataset

<table>
<thead>
<tr>
<th>Query</th>
<th>join height</th>
<th>selectivity</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GF 1P</td>
<td>1</td>
<td>0.00498</td>
<td>Select titles of articles without author</td>
</tr>
<tr>
<td>GF 1Q</td>
<td>1</td>
<td>0.00190</td>
<td>Select names of authors without any publication</td>
</tr>
<tr>
<td>GF 1R</td>
<td>1</td>
<td>0.00001</td>
<td>Select names of authors that wrote all publication types except books</td>
</tr>
<tr>
<td>GF 3P</td>
<td>3</td>
<td>0.29406</td>
<td>Select titles of books only written by authors that never wrote articles</td>
</tr>
<tr>
<td>GF 3Q</td>
<td>3</td>
<td>0.60255</td>
<td>Select titles of books only written by authors that also wrote articles</td>
</tr>
</tbody>
</table>

Table 4.8 lists the guarded fragment queries that are tested on the DBLP dataset. Figure 4.27 shows
4.5. K-BISIMULATION STRUCTURAL INDEXING FOR DBLP

![Figure 4.26: Running time acceleration for FACQ queries using k-bisimulation partitioning on DBLP](image1)

Figure 4.26: Running time acceleration for FACQ queries using k-bisimulation partitioning on DBLP

![Figure 4.27: Running time acceleration for guarded fragment queries using k-bisimulation partitioning on DBLP](image2)

Figure 4.27: Running time acceleration for guarded fragment queries using k-bisimulation partitioning on DBLP
the acceleration factor for the guarded fragment queries that were tested on the DBLP dataset. Queries GF 1P, 1Q and 1R show an acceleration of at least 3.1 times for $k \leq 4$. Queries GF 3P and GF 3Q have a similar characteristic as the FACQ queries of height 3. They can be expressed with less joins on the original database and therefore these queries do not show an acceleration under $I$-bisimulation. Their maximum acceleration of respectively 61 and 517 times is achieved when a structural index for $k = 4$ is used, but the performance is comparable to $k = 3$. For $k > 4$, the acceleration drops for all queries. Queries GF 3P and 3Q are very similar, but GF 3Q shows a higher acceleration factor of at most 505 times.

4.5.6 Discussion

In this section, we evaluated $k$-bisimulation structural indexing using a real world dataset. We accelerated several FACQ and guarded fragment queries. The number of partition blocks in the $k$-bisimulation partitioning grows much faster than with TPC-H. A reason can be that DBLP contains more social-like data. As a result, there are lower acceleration factors possible and it is harder to find a $k$ value that gives a good acceleration factor for all tested queries. For $k = 1$ or $k > 4$, some queries are slowed down. The best results are found under $I$-bisimulation, which gives at least 2.9 times better performance for all tested queries, and up to 505 times for one particular query.

4.6 Workload-driven structural indexing

In this section, we show some experiments for some alternatives for $k$-bisimulation structural indexing. We will provide small experiments for some of the approaches for workload-driven structural indexing described in Section 3.7. Further, we will partition a relation based on query results and build DAG-based structural indices.

4.6.1 Relation partitioning on query result

In PostgreSQL it is possible combine the functionality of an update statement with a select statement. It can perform an update statement, and at the same time return the result to the user. We can use this feature to tag the tuples that result from a query in the original relation. Listing 4.7 shows an original semijoin query and Listing 4.8 shows a tagging version of the same query following the approach described in Subsection 3.7.2. The tagging version of the query uses the `RETURNING` function in PostgreSQL to tag the resulting tuples in the original relation. We use a `block_id` attribute to store the tags. It is possible to assign each bit to a different query result. Listing 4.9 shows the accelerated version of the query. It outputs the desired tuples based on the tags.

```sql
SELECT customer.c_name
FROM customer
WHERE EXISTS
  ( SELECT 1
    FROM orders
    WHERE o_custkey = customer.c_custkey
  )

Listing 4.7: Original semijoin query

UPDATE customer
SET block_id = 1
WHERE EXISTS
  ( SELECT 1
    FROM orders
    WHERE o_custkey = customer.c_custkey
  )
RETURNING customer.c_name

Listing 4.8: Tagging semijoin query

SELECT customer.c_name
FROM customer
WHERE block_id = 1

Listing 4.9: Accelerated semijoin query
The running times are as follows:

original query 1068 ms

tagging query 3091 ms

accelerated query 99 ms

We can observe that the tagging query is approximately three times slower than the original query. When the same query will be executed more than three times, it is faster to use the tagging query and then use then use accelerated query. The total running time will be lower. However, any update to the relations will invalidate the tags.

Our small example shows that it is feasible to tag tuples based on the query result. Several improvements are possible. For example, the DBMS can be modified to first return the query result to the client and then process the update. We can also partially answer a more specific query by pruning information from tags.

4.6.2 DAG-based structural indexing

This subsection will provide an experimental analysis for our approach from Subsection 3.7.3. We will describe a set of queries and then try to design a structural index specifically to this workload.

Table 4.9 lists some queries on the TPC-H dataset. Those queries have common join conditions. Query FACQ 2C and query GF 2B access the nation, customer and supplier relations. Similarly, queries FACQ 2A, FACQ 2B and GF 4A access the part and partsupp relations.

Table 4.9: Example queries for DAG-based structural indexing on TPC-H

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FACQ 2C</td>
<td>Select all regions having a nation with both customers and suppliers</td>
</tr>
<tr>
<td>GF 2B</td>
<td>Select all regions not having a nation with customers but no suppliers</td>
</tr>
<tr>
<td>FACQ 2A</td>
<td>Select names of suppliers having a lineitem</td>
</tr>
<tr>
<td>FACQ 2B</td>
<td>Select names of parts having a lineitem</td>
</tr>
<tr>
<td>GF 4A</td>
<td>Select all suppliers that sell parts that are offered but not sold by other suppliers</td>
</tr>
</tbody>
</table>

Figure 4.28 visualizes each query program using a tree. A suitable partitioning order can be derived from these trees. DAGs are a useful data structure to specify partitioning order constraints for structural indices. Its structure is similar to the visualized query programs that it covers. At the same time, it allows for concurrent computation of partitionings.

Figure 4.29 proposes two DAGs that would cover the example queries in Table 4.9. Figure 4.30 shows the DAGs mapped on the schema. The DAG 1 is shown in red and covers queries FACQ 2C and GF 2B. Similarly, DAG 2 is shown in blue and covers queries FACQ 2A, FACQ 2B and GF 4A.
CHAPTER 4. EXPERIMENTAL STUDY

Figure 4.28: Query program tree

(a) FACQ 2C

(b) GF 2B

(c) FACQ 2A

(d) FACQ 2B

(e) GF 4A

Figure 4.29: DAG bisimulation visualization

(a) DAG 1

(b) DAG 2
4.6. WORKLOAD-DRIVEN STRUCTURAL INDEXING

DAG structural indexing is implemented in C++ and SQL. The topological sort is performed using the Boost graph library. The signature generation is offloaded to SQL. Listing 4.10 gives an example signature generation for the nation relation.

```sql
SELECT n_nationkey, customer_signatures.signature, supplier_signatures.signature
FROM nation
LEFT OUTER JOIN
    (SELECT c_nationkey, array_agg(DISTINCT c_block_id) signature
        FROM customer
        GROUP BY c_nationkey
    ) customer_signatures
ON c_nationkey = n_nationkey
LEFT OUTER JOIN
    (SELECT s_nationkey, array_agg(DISTINCT s_block_id) signature
        FROM supplier
        GROUP BY s_nationkey
    ) supplier_signatures
ON s_nationkey = n_nationkey
```

Listing 4.10: Nation signature generation

We evaluated the index generation and usage for DAG 2 and compared the results with $k$-bisimulation structural indexing. The generation of the DAG-based structural index took 92 seconds, which is much faster than the 27 minutes that would be required for a 4-bisimulation structural index that can also answer those queries. The DAG-based structural index gave similar query performance as for 4-bisimulation structural indexing. Because of the lower index generation times, we can conclude that DAG-based structural indices are usable when the queries are known upfront.
Chapter 5

Conclusions

This work provides a first experimental evaluation of the creation and use of structural indices for relational databases. We demonstrated that it is possible to extend structural indexing from graph databases to relational databases. We visualized which information can be extracted from these indices and we analyzed the performance or our approach on both synthetic and real-world datasets.

5.1 Contributions

We showed how the structure of a relational database can be represented as a graph. We demonstrated how primary keys and foreign keys can be used to create nodes and edges. Challenges with mapping primary key values to node identifiers have been identified as well. We presented two methods to accelerate join processing: by evaluating the joins on the structural index, or partially evaluate queries on the original database by pruning information from the index.

We evaluated our $k$-bisimulation structural indexing approach with both synthetic and real-world datasets. The output of the $k$-bisimulation partitioning was visualized and analyzed. Then, we analyzed which query and data properties have an influence on the query acceleration factors that can be achieved. For synthetic TPC-H dataset, the reduced graphs are significantly smaller than the original dataset for values of $k \leq 8$, while it can accelerate more complex queries for values of $k > 3$. For the real-world DBLP dataset, the reduced graph is significantly smaller for $k \leq 4$, while there are still practical queries for $k = 5$. We demonstrated that structural indexing can significantly accelerate semijoin and antijoin queries. In certain cases, the query running time can be up to 500 times faster. Typically, this happens when the number of $k$-bisimulation iterations is equal to the query height. Therefore, it is not possible for some datasets to create an index that is gives good acceleration for all queries.

For queries that cannot be evaluated by using $k$-bisimulation structural indexing, we showed how the values of attributes can be embedded into structural indices to create a hybrid structural index. Using a hybrid structural index, we demonstrated that it is possible to accelerate semijoin or antijoin queries with nested value-based selection conditions. Hybrid structural indexing with attribute information gave a small performance boost for this type of queries. This would not be possible using regular $k$-bisimulation structural indexing.

Our experiments showed that structural indices are not very useful for equijoins. It was not possible to produce any significant acceleration for equijoin queries for our datasets. It is hard to beat the performance for equijoin processing that can be achieved using traditional B-tree indices. The reasons for this result have been discussed as well.

Based on our observations, we can conclude that structural indexing can be extended to relational databases to accelerate query processing. Significant acceleration factors can be achieved for both FACQ and guarded fragment queries. These queries can be accelerated with several orders of magnitude. At the same time, structural indices require less disk space than traditional B-tree indices. The acceleration factor highly depends on the sizes of the joined relations, the number of join operations and the type of data. By choosing a suitable $k$ value, we can build a structural index that gives a good acceleration for all queries.
5.2 Limitations

All queries that showed an acceleration are queries that contain semijoin or antijoin operations. These queries are also known as tuple selecting queries. Our structural indexing approach did not give a significant acceleration for equijoin queries, known as tuple generating queries.

The implementation for $k$-bisimulation has some limitations. We converted all node identifiers to 32-bit integers to be able to use an existing $k$-bisimulation partitioning algorithm. After $k$-bisimulation, the identifiers must be converted back to the original primary keys to be able to answer queries. The conversion between the original primary keys and 32-bit identifiers may impact the index generation performance. Changing the $k$-bisimulation implementation to use multi-attribute identifiers would allow direct conversion between primary key values and node identifiers. This would improve the index generation performance and allows to create structural indices for database that contain more than $2^{32}$ tuples.

5.3 Future work

Additional research can be done in several directions. We showed how queries can be accelerated by rewriting them to queries that use a structural index. A challenge is to integrate this technique in a query planner that automatically identifies query parts that can be evaluated using the structural index. The query planner then needs to estimate the costs for evaluating queries on a structural index.

Another challenge is to maintain the structural index after updates to the original data. The updates should be efficiently applied on the index without regenerating the complete index. In the worst-case scenario, some updates would be more complex than regenerating the whole structural index.
Bibliography


Appendix A

Structural Index schema for PostgreSQL

-- Reduced graph nodes (partition blocks) storage:
-- Substitute "node" for every relation in the database
CREATE TABLE structural_index.node
(
  id integer NOT NULL,
  size integer NOT NULL,
  CONSTRAINT node_pkey PRIMARY KEY (id)
)

-- Reduced graph edges storage:
-- Substitute "edge" for every foreign key constraint name in the database
CREATE TABLE structural_index.edge
(
  from_id integer NOT NULL,
  to_id integer NOT NULL,
  CONSTRAINT edge_forward_index PRIMARY KEY (from_id, to_id)
)

-- Reduced graph edges B-tree indexing:
-- The edge_forward_index B-tree is automatically added by the primary key
-- Substitute "edge" for each relation name in the database
CREATE INDEX edge_backward_index
ON edge
USING btree
  (to_id, from_id);

-- Partitioning storage in block_id attribute:
-- Substitute "relation" for each relation name in the database
ALTER TABLE relation ADD COLUMN block_id integer NOT NULL;
Appendix B

TPC-H Schema for PostgreSQL

CREATE TABLE nation
(
  n_nationkey serial NOT NULL,
  n_name character(25),
  n_regionkey bigint NOT NULL,
  n_comment character varying(152),
  CONSTRAINT nation_pkey PRIMARY KEY (n_nationkey),
  CONSTRAINT nation_n_regionkey_fkey FOREIGN KEY (n_regionkey)
    REFERENCES region (r_regionkey) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
)

CREATE TABLE nation
(
  n_nationkey serial NOT NULL,
  n_name character(25),
  n_regionkey bigint NOT NULL,
  n_comment character varying(152),
  CONSTRAINT nation_pkey PRIMARY KEY (n_nationkey),
  CONSTRAINT nation_n_regionkey_fkey FOREIGN KEY (n_regionkey)
    REFERENCES region (r_regionkey) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
)

CREATE TABLE customer
(
  c_custkey serial NOT NULL,
  c_name character varying(25),
  c_address character varying(40),
  c_nationkey bigint NOT NULL,
  c_phone character(15),
  c_acctbal numeric,
  c_mktsegment character(10),
  c_comment character varying(117),
  CONSTRAINT customer_pkey PRIMARY KEY (c_custkey),
  CONSTRAINT customer_c_nationkey_fkey FOREIGN KEY (c_nationkey)
    REFERENCES nation (n_nationkey) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
)

CREATE TABLE part
(
  p_partkey serial NOT NULL,
  p_name character varying(55),
  p_mfgr character(25),
  p_brand character(10),
  p_type character varying(25),
  p_size integer,
  p_service character varying(23),
  p_type character varying(25),
  p_size integer,
  p_service character varying(23),
  CONSTRAINT part_pkey PRIMARY KEY (p_partkey),
  CONSTRAINT part_p_mfgr_fkey FOREIGN KEY (p_mfgr)
    REFERENCES supplier (s_mfgr) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
)
p_container character(10),
p_retailprice numeric,
p_comment character varying(23),
CONSTRAINT part_pkey PRIMARY KEY (p_partkey)
)

CREATE TABLE supplier
(
s_suppkey serial NOT NULL,
s_name character(25),
s_address character varying(40),
s_nationkey bigint NOT NULL,
s_phone character(15),
s_acctbal numeric,
s_comment character varying(101),
CONSTRAINT supplier_pkey PRIMARY KEY (s_suppkey),
CONSTRAINT supplier_s_nationkey_fkey FOREIGN KEY (s_nationkey)
    REFERENCES nation (n_nationkey) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
)

CREATE TABLE orders
(
o_orderkey serial NOT NULL,
o_custkey bigint NOT NULL,
o_orderstatus character(1),
o_totalprice numeric,
o_orderdate date,
o_orderpriority character(15),
o_clerk character(15),
o_shippriority integer,
o_comment character varying(79),
CONSTRAINT orders_pkey PRIMARY KEY (o_orderkey),
CONSTRAINT orders_o_custkey_fkey FOREIGN KEY (o_custkey)
    REFERENCES customer (c_custkey) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
)

CREATE TABLE partsupp
(
ps_partkey bigint NOT NULL,
ps_suppkey bigint NOT NULL,
ps_availqty integer,
ps_supplycost numeric,
ps_comment character varying(199),
CONSTRAINT partsupp_pkey PRIMARY KEY (ps_partkey, ps_suppkey),
CONSTRAINT partsupp_ps_partkey_fkey FOREIGN KEY (ps_partkey)
    REFERENCES part (p_partkey) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION,
CONSTRAINT partsupp_ps_suppkey_fkey FOREIGN KEY (ps_suppkey)
    REFERENCES supplier (s_suppkey) MATCH SIMPLE
    ON UPDATE NO ACTION ON DELETE NO ACTION
)

CREATE TABLE lineitem
(
l_orderkey bigint NOT NULL,
l_partkey bigint NOT NULL,
l_suppkey bigint NOT NULL,
l_linenumber integer NOT NULL,
l_quantity numeric,
l_extendedprice numeric,
l_discount numeric,
l_tax numeric,
l_returnflag character(1),
l_linestatus character(1),
l_shipdate date,
l_commitdate date,
l_receiptdate date,
l_shipinstruct character(25),
l_shipmode character(10),
l_comment character varying(44),
CONSTRAINT lineitem_pkey PRIMARY KEY (l_orderkey, l_linenumber),
CONSTRAINT lineitem_l_orderkey_fkey FOREIGN KEY (l_orderkey)
  REFERENCES orders (o_orderkey) MATCH SIMPLE
  ON UPDATE NO ACTION ON DELETE NO ACTION,
CONSTRAINT lineitem_l_partkey_fkey FOREIGN KEY (l_partkey, l_suppkey)
  REFERENCES partsupp (ps_partkey, ps_suppkey) MATCH SIMPLE
  ON UPDATE NO ACTION ON DELETE NO ACTION
)

-- B-tree indices on referencing foreign key attributes to accelerate traditional joins
CREATE INDEX customer_nationkey_fkey_index
  ON customer
  USING btree
  (c_nationkey);

CREATE INDEX lineitem_orderkey_fkey_index
  ON lineitem
  USING btree
  (l_orderkey);

CREATE INDEX lineitem_partsupp_fkey_index
  ON lineitem
  USING btree
  (l_partkey, l_suppkey);

CREATE INDEX nation_regionkey_fkey_index
  ON nation
  USING btree
  (n_regionkey);

CREATE INDEX orders_custkey_fkey_index
  ON orders
  USING btree
  (o_custkey);

CREATE INDEX partsupp_partkey_fkey_index
  ON partsupp
  USING btree
  (ps_partkey);

CREATE INDEX partsupp_suppkey_fkey_index
  ON partsupp
  USING btree
  (ps_suppkey);

CREATE INDEX supplier_nationkey_fkey_index
  ON supplier
  USING btree
  (s_nationkey);
Appendix C

TPC-H Queries

C.1 Freely Acyclic Conjunctive Queries

Query FACQ1A: select customer names having order

```sql
SELECT c_name
FROM customer
WHERE block_id IN
( SELECT to_id
  FROM structural_index.orders_o_custkey_fkey
);
```

Query FACQ1B: select partsupp supplycost having lineitem

```sql
SELECT partsupp.ps_partkey, partsupp.ps_suppkey, partsupp.ps_supplycost
FROM partsupp
WHERE block_id IN
( SELECT DISTINCT to_id
  FROM structural_index.lineitem_l_partkey_fkey
);
```
C.1. FREELY ACYCLIC CONJUNCTIVE QUERIES

Original

```
SELECT supplier.s_name
FROM supplier
WHERE EXISTS
  (  
    SELECT 1
    FROM partsupp
    WHERE supplier.s_suppkey = partsupp.ps_suppkey
    AND EXISTS
      (  
        SELECT 1
        FROM lineitem
        WHERE partsupp.ps_partkey = lineitem.l_partkey
        AND partsupp.ps_suppkey = lineitem.l_suppkey
      )
  )
```

```
SELECT s_name
FROM supplier
WHERE EXISTS
  (  
    SELECT 1
    FROM partsupp
    WHERE partsupp.ps_suppkey = supplier.s_suppkey
    AND EXISTS
      (  
        SELECT 1
        FROM structural_index.lineitem_l_partkey_fkey
        WHERE partsupp.block_id = lineitem_l_partkey_fkey.to_id
      )
  )
```

```
SELECT s_name
FROM supplier
WHERE block_id IN
  (  
    SELECT edge1.to_id
    FROM structural_index.partsupp_ps_suppkey_fkey as edge1
    INNER JOIN structural_index.lineitem_l_partkey_fkey as edge2
    ON edge1.from_id = edge2.to_id
  )
```

Query FACQ2A: select supplier names having lineitem
APPENDIX C. TPC-H QUERIES

SELECT part.p_partkey, part.p_name
FROM part
WHERE EXISTS
  (  SELECT 1
  FROM partsupp
  WHERE part.p_partkey = partsupp.ps_partkey
  AND EXISTS
    (  SELECT 1
    FROM lineitem
    WHERE partsupp.ps_partkey = lineitem.l_partkey
    AND partsupp.ps_suppkey = lineitem.l_suppkey
  )
);

SELECT p_partkey, p_name
FROM part
WHERE block_id IN
  (  SELECT edge1.to_id
  FROM structural_index.partsupp_ps_partkey_fkey as edge1
  WHERE EXISTS
    (  SELECT 1
    FROM structural_index.lineitem_l_partkey_fkey as edge2
    WHERE edge1.from_id = edge2.to_id
    )
  )
;

Query FACQ2B: select part names having lineitem
C.1. FREELY ACYCLIC CONJUNCTIVE QUERIES

SELECT r_name FROM region
WHERE EXISTS
  ( SELECT 1 FROM nation WHERE region.r_regionkey = nation.n_regionkey
  AND EXISTS
    ( SELECT 1 FROM customer WHERE nation.n_nationkey = customer.c_nationkey
    AND EXISTS
      ( SELECT 1 FROM orders WHERE customer.c_custkey = orders.o_custkey
      AND EXISTS
        ( SELECT 1 FROM lineitem
            WHERE orders.o_orderkey = lineitem.l_orderkey
        ) ) ) )

SELECT r_name FROM region
WHERE EXISTS
  ( SELECT 1 FROM nation WHERE region.r_regionkey = nation.n_regionkey
  AND EXISTS
    ( SELECT 1 FROM customer WHERE nation.n_nationkey = customer.c_nationkey
    AND EXISTS
      ( SELECT 1 FROM orders
          WHERE orders.block_id IN
            ( SELECT lineitem_l_orderkey_fkey.to_id
                FROM structural_index.lineitem_l_orderkey_fkey
            ) ) ) )

SELECT r_name FROM region
WHERE EXISTS
  ( SELECT 1 FROM nation WHERE region.r_regionkey = nation.n_regionkey
  AND EXISTS
    ( SELECT 1 FROM structural_index.customer_c_nationkey_fkey
        WHERE customer_c_nationkey_fkey.to_id = nation.block_id
    AND EXISTS
      ( SELECT 1 FROM structural_index.orders_o_custkey_fkey
          WHERE customer_c_nationkey_fkey.from_id = orders_o_custkey_fkey.to_id
      AND EXISTS
        ( SELECT 1 FROM structural_index.lineitem_l_orderkey_fkey
            WHERE orders_o_custkey_fkey.from_id = lineitem_l_orderkey_fkey.to_id
        ) ) ) )

SELECT r_name FROM region
WHERE EXISTS
  ( SELECT nation_n_regionkey_fkey.to_id FROM structural_index.nation_n_regionkey_fkey
    WHERE nation_n_regionkey_fkey.from_id = nation_n_regionkey_fkey.to_id
  AND EXISTS
    ( SELECT 1 FROM structural_index.customer_c_nationkey_fkey
        WHERE nation_n_regionkey_fkey.from_id = customer_c_nationkey_fkey.to_id
    AND EXISTS
      ( SELECT 1 FROM structural_index.orders_o_custkey_fkey
          WHERE customer_c_nationkey_fkey.from_id = orders_o_custkey_fkey.to_id
      AND EXISTS
        ( SELECT 1 FROM structural_index.lineitem_l_orderkey_fkey
            WHERE orders_o_custkey_fkey.from_id = lineitem_l_orderkey_fkey.to_id
        ) ) )

Query FACQ4A: select region having nation customer orders lineitem
C.2 Guarded Fragment Queries

```sql
Original

SELECT partsupp.ps_partkey, partsupp.ps_suppkey, partsupp.ps_comment
FROM partsupp
WHERE NOT EXISTS
  ( SELECT 1
      FROM lineitem
      WHERE l_partkey = ps_partkey
      AND l_suppkey = ps_suppkey
  );

k > 0

SELECT partsupp.ps_partkey, partsupp.ps_suppkey, partsupp.ps_comment
FROM partsupp
WHERE block_id IN
  ( SELECT DISTINCT partsupp.id
      FROM structural_index.partsupp
      WHERE NOT EXISTS
        ( SELECT to_id
           FROM structural_index.lineitem_l_partkey_fkey
           WHERE partsupp.id = to_id
        )
  );

Query GF1A: select partsupp comment not having lineitem

Original

SELECT c_name
FROM customer
WHERE NOT EXISTS
  ( SELECT 1
      FROM orders
      WHERE o_custkey = c_custkey
  );

k > 0

SELECT c_name
FROM customer
WHERE block_id IN
  ( SELECT structural_index.customer.id
      FROM structural_index.customer
      WHERE NOT EXISTS
        ( SELECT 1
           FROM structural_index.orders_o_custkey_fkey
           WHERE orders_o_custkey_fkey.to_id = customer.id
        )
  );

Query GF1B: select customers not having order
SELECT p_partkey, p_name
FROM part
WHERE p_partkey IN
    (SELECT ps_partkey
     FROM partsupp
     WHERE NOT EXISTS
         (SELECT 1
          FROM lineitem
          WHERE lineitem.l_partkey = partsupp.ps_partkey
          AND lineitem.l_suppkey = partsupp.ps_suppkey
         )
    );

SELECT p_partkey, p_name
FROM part
WHERE EXISTS
    (SELECT 1
     FROM partsupp
     WHERE part.p_partkey = partsupp.ps_partkey
     AND NOT EXISTS
         (SELECT 1
          FROM structural_index.lineitem_l_partkey_fkey
          WHERE partsupp.block_id = lineitem_l_partkey_fkey.to_id
         )
    )

SELECT p_partkey, p_name
FROM part
WHERE block_id IN
    (SELECT partsupp_ps_partkey_fkey.to_id
     FROM structural_index.partsupp_ps_partkey_fkey
     WHERE NOT EXISTS
         (SELECT 1
          FROM structural_index.lineitem_l_partkey_fkey
          WHERE partsupp_ps_partkey_fkey.from_id = lineitem_l_partkey_fkey.to_id
         )
    );

Query GF2A: select parts having partsupp without lineitem
SELECT s_suppkey
FROM supplier
WHERE EXISTS
( SELECT 1
  FROM partsupp
  WHERE ps_suppkey = s_suppkey
  AND EXISTS
  ( SELECT 1
    FROM lineitem
    WHERE l_suppkey = ps_suppkey
    AND l_partkey = ps_partkey
  )
  AND EXISTS
  ( SELECT 1
    FROM part
    WHERE p_partkey = ps_partkey
    AND EXISTS
    ( SELECT 1
      FROM partsupp partsupp2
      WHERE partsupp2.ps_partkey = part.p_partkey
      AND NOT EXISTS
      ( SELECT 1
        FROM lineitem lineitem2
        WHERE lineitem2.l_partkey = partsupp2.ps_partkey
        AND lineitem2.l_suppkey = partsupp2.ps_suppkey
      )
    )
  )
)

SELECT s_suppkey FROM supplier
WHERE EXISTS
( SELECT 1 FROM partsupp
  WHERE ps_suppkey = s_suppkey
  AND EXISTS
  ( SELECT 1 FROM structural_index.lineitem_l_partkey_fkey
    WHERE lineitem_l_partkey_fkey.to_id = partsupp.block_id
  )
  AND EXISTS
  ( SELECT 1 FROM part
    WHERE p_partkey = ps_partkey
    AND EXISTS
    ( SELECT 1 FROM partsupp partsupp2
      WHERE partsupp2.ps_partkey = part.p_partkey
      AND NOT EXISTS
      ( SELECT 1 FROM structural_index.lineitem_l_partkey_fkey lineitem_l_partkey_fkey2
        WHERE lineitem_l_partkey_fkey2.to_id = partsupp2.block_id
      )
    )
  )
)

Query GF4A: Select all suppliers that sell parts that are offered but not sold by other suppliers
SELECT s_suppkey FROM supplier
WHERE EXISTS
( SELECT 1 FROM partsupp
    WHERE ps_suppkey = s_suppkey
    AND EXISTS
        ( SELECT 1 FROM structural_index.lineitem_l_partkey_fkey
            WHERE lineitem_l_partkey_fkey.to_id = partsupp.block_id
        )
    AND EXISTS
        ( SELECT 1 FROM part
            WHERE p_partkey = ps_partkey
            AND EXISTS
                ( SELECT 1 FROM structural_index.partsupp_ps_partkey_fkey
                    WHERE partsupp_ps_partkey_fkey.to_id = part.block_id
                )
            AND EXISTS
                ( SELECT 1 FROM structural_index.partsupp_ps_partkey_fkey partsupp_ps_partkey_fkey2
                    WHERE partsupp_ps_partkey_fkey2.to_id = partsupp.block_id
                    AND NOT EXISTS
                        ( SELECT 1 FROM structural_index.lineitem_l_partkey_fkey lineitem_l_partkey_fkey2
                            WHERE lineitem_l_partkey_fkey2.to_id = partsupp_ps_partkey_fkey2.from_id
                        )
                )
        )
) ) )

SELECT s_suppkey FROM supplier
WHERE EXISTS
( SELECT 1 FROM partsupp
    WHERE ps_suppkey = s_suppkey
    AND EXISTS
        ( SELECT 1 FROM structural_index.lineitem_l_partkey_fkey
            WHERE lineitem_l_partkey_fkey.to_id = partsupp.block_id
        )
    AND EXISTS
        ( SELECT 1 FROM structural_index.partsupp_ps_partkey_fkey
            WHERE partsupp_ps_partkey_fkey.to_id = partsupp.block_id
            AND EXISTS
                ( SELECT 1 FROM structural_index.partsupp_ps_partkey_fkey partsupp_ps_partkey_fkey2
                    WHERE partsupp_ps_partkey_fkey2.to_id = partsupp_ps_partkey_fkey.to_id
                    AND EXISTS
                        ( SELECT 1 FROM structural_index.lineitem_l_partkey_fkey lineitem_l_partkey_fkey2
                            WHERE lineitem_l_partkey_fkey2.to_id = partsupp_ps_partkey_fkey2.from_id
                        )
                )
            AND EXISTS
                ( SELECT 1 FROM structural_index.partsupp_ps_partkey_fkey partsupp_ps_partkey_fkey2
                    WHERE partsupp_ps_partkey_fkey2.to_id = partsupp_ps_partkey_fkey.to_id
                    AND EXISTS
                        ( SELECT 1 FROM structural_index.lineitem_l_partkey_fkey lineitem_l_partkey_fkey2
                            WHERE lineitem_l_partkey_fkey2.to_id = partsupp_ps_partkey_fkey2.from_id
                        )
                )
        )
) )

SELECT s_suppkey FROM supplier
WHERE EXISTS
( SELECT 1 FROM structural_index.partsupp_ps_suppkey_fkey
    WHERE partsupp_ps_suppkey_fkey.to_id = supplier.block_id
    AND EXISTS
        ( SELECT 1 FROM structural_index.lineitem_l_partkey_fkey
            WHERE lineitem_l_partkey_fkey.to_id = partsupp_ps_suppkey_fkey.from_id
        )
    AND EXISTS
        ( SELECT 1 FROM structural_index.lineitem_l_partkey_fkey
            WHERE lineitem_l_partkey_fkey.to_id = partsupp_ps_suppkey_fkey.from_id
            AND EXISTS
                ( SELECT 1 FROM structural_index.lineitem_l_partkey_fkey lineitem_l_partkey_fkey2
                    WHERE lineitem_l_partkey_fkey2.to_id = partsupp_ps_suppkey_fkey2.from_id
                )
            AND EXISTS
                ( SELECT 1 FROM structural_index.lineitem_l_partkey_fkey lineitem_l_partkey_fkey2
                    WHERE lineitem_l_partkey_fkey2.to_id = partsupp_ps_suppkey_fkey2.from_id
                )
        )
) )

Query GF4A: (continued)
C.3 Value-based Selective Queries

```sql
Original

SELECT c_name
FROM customer
WHERE c_acctbal < -999.95
AND EXISTS
  ( SELECT 1
    FROM orders
    WHERE customer.c_custkey = orders.o_custkey
  )

SELECT c_name
FROM customer
WHERE c_acctbal < -999.5
AND EXISTS
  ( SELECT 1
    FROM structural_index.orders_o_custkey_fkey
    WHERE to_id = customer.block_id
  )

Query VS1A: select customer names having order where acctbal below value

k > 1

SELECT part.p_partkey, part.p_name
FROM part
WHERE EXISTS
  ( SELECT 1
    FROM partsupp
    WHERE part.p_partkey = partsupp.ps_partkey
    AND EXISTS
      ( SELECT 1
        FROM lineitem
        WHERE partsupp.ps_partkey = lineitem.l_partkey
        AND partsupp.ps_suppkey = lineitem.l_suppkey
      )
  )
AND p_size <= 1

SELECT p_partkey, p_name
FROM part
WHERE block_id IN
  ( SELECT edge1.to_id
    FROM structural_index.partsupp_ps_partkey_fkey as edge1
    WHERE EXISTS
      ( SELECT 1
        FROM structural_index.lineitem_l_partkey_fkey as edge2
        WHERE edge1.from_id = edge2.to_id
      )
  )
AND p_size <= 1

Query VS2A: select part names having lineitem variable size

k > 0
```
C.4 Hybrid Queries

 ORIGINAL

\[
\begin{align*}
\text{SELECT} & \quad \text{c\_name} \\
\text{FROM} & \quad \text{customer} \\
\text{WHERE} & \quad \text{EXISTS} \\
( & \quad \text{SELECT 1} \\
& \quad \text{FROM} \quad \text{orders} \\
& \quad \text{WHERE} \quad \text{orders.o\_custkey} = \text{customer.c\_custkey} \\
& \quad \text{AND} \quad \text{o\_orderstatus} = \text{’P’} \\
\text{)};
\end{align*}
\]

\[
\begin{align*}
\text{SELECT} & \quad \text{c\_name} \quad \text{FROM} \quad \text{customer} \\
\text{WHERE} & \quad \text{EXISTS} \\
( & \quad \text{SELECT 1} \\
& \quad \text{FROM} \quad \text{structural\_index.orders.o\_custkey\_fkey} \\
& \quad \text{WHERE} \quad \text{order.o\_custkey\_fkey.to\_id} = \text{customer.block\_id} \\
& \quad \text{AND} \quad \text{EXISTS} \\
& \quad \quad ( & \quad \text{SELECT 1} \\
& \quad \quad \text{FROM} \quad \text{structural\_index.orders} \\
& \quad \quad \text{WHERE} \quad \text{orders.id} = \text{orders.o\_custkey\_fkey.from\_id} \\
& \quad \quad \text{AND} \quad \text{o\_orderstatus} = \text{’P’} \\
& \quad \quad ) \\
\text{)}
\end{align*}
\]

Query HYBRID 1A: select names of customers that have a pending order

\[
\begin{align*}
\text{SELECT} & \quad \text{c\_name} \\
\text{FROM} & \quad \text{customer} \\
\text{WHERE} & \quad \text{EXISTS} \\
( & \quad \text{SELECT 1} \\
& \quad \text{FROM} \quad \text{orders} \\
& \quad \text{WHERE} \quad \text{orders.o\_custkey} = \text{customer.c\_custkey} \\
& \quad \text{AND} \quad \text{o\_orderstatus} = \text{’F’} \\
\text{)};
\end{align*}
\]

\[
\begin{align*}
\text{SELECT} & \quad \text{c\_name} \quad \text{FROM} \quad \text{customer} \\
\text{WHERE} & \quad \text{EXISTS} \\
( & \quad \text{SELECT 1} \\
& \quad \text{FROM} \quad \text{structural\_index.orders.o\_custkey\_fkey} \\
& \quad \text{WHERE} \quad \text{order.o\_custkey\_fkey.to\_id} = \text{customer.block\_id} \\
& \quad \text{AND} \quad \text{EXISTS} \\
& \quad \quad ( & \quad \text{SELECT 1} \\
& \quad \quad \text{FROM} \quad \text{structural\_index.orders} \\
& \quad \quad \text{WHERE} \quad \text{orders.id} = \text{orders.o\_custkey\_fkey.from\_id} \\
& \quad \quad \text{AND} \quad \text{o\_orderstatus} = \text{’F’} \\
& \quad \quad ) \\
\text{)}
\end{align*}
\]

Query HYBRID 1B: select names of customers that have a finished order
APPENDIX C. TPC-H QUERIES

Original

```sql
SELECT n_name
FROM nation
WHERE EXISTS
  (
    SELECT 1
    FROM customer
    WHERE customer.c_nationkey = nation.n_nationkey
    AND EXISTS
      (
        SELECT 1
        FROM orders
        WHERE orders.o_custkey = customer.c_custkey
        AND o_orderstatus = 'P'
      )
  );
```

$k > 0$

```sql
SELECT n_name
FROM nation
WHERE EXISTS
  (
    SELECT 1
    FROM structural_index.customer_c_nationkey_fkey
    WHERE customer_c_nationkey_fkey.to_id = nation.block_id
    AND EXISTS
      (
        SELECT 1
        FROM structural_index.orders_o_custkey_fkey
        WHERE orders_o_custkey_fkey.to_id = customer_c_nationkey_fkey.from_id
        AND EXISTS
          (
            SELECT 1
            FROM structural_index.orders
            WHERE orders.id = orders_o_custkey_fkey.from_id
            AND o_orderstatus = 'P'
          )
      )
  );
```

Query HYBRID 2A: select names of nations having a customer with a pending order
C.4. HYBRID QUERIES

Original

```sql
SELECT p_name
FROM part
WHERE EXISTS
  (  
  SELECT 1
  FROM partsupp
  WHERE partsupp.ps_partkey = part.p_partkey
  AND EXISTS
    ( SELECT 1
      FROM lineitem
      WHERE lineitem.l_partkey = partsupp.ps_partkey
      AND lineitem.l_suppkey = partsupp.ps_suppkey
      AND EXISTS
        ( SELECT 1
          FROM orders
          WHERE orders.o_orderkey = lineitem.l_orderkey
          AND orders.o_orderstatus = 'P'
        )
    )
  );
```

```sql
SELECT p_name
FROM part
WHERE EXISTS
  (  
  SELECT 1
  FROM structural_index.partsupp_ps_partkey_fkey
  WHERE partsupp_ps_partkey_fkey.to_id = part.block_id
  AND EXISTS
    ( SELECT 1
      FROM structural_index.lineitem_l_partkey_fkey
      WHERE lineitem_l_partkey_fkey.to_id = partsupp_ps_partkey_fkey.from_id
      AND EXISTS
        ( SELECT 1
          FROM structural_index.lineitem_l_orderkey_fkey
          WHERE lineitem_l_orderkey_fkey.from_id = lineitem_l_partkey_fkey.from_id
          AND EXISTS
            ( SELECT 1
              FROM structural_index.orders
              WHERE orders.id = lineitem_l_orderkey_fkey.to_id
              AND o_orderstatus = 'P'
            )
        )
    )
  );
```

Query HYBRID 3A: select names of parts having pending order through partsupp and lineitem
Appendix D

DBLP Schema for PostgreSQL

CREATE TABLE article
(
    pubid integer NOT NULL DEFAULT nextval('publication_pubid_seq '::regclass),
    pubkey text,
    title text,
    year integer,
    journal text,
    month text,
    volume text,
    "number" text,
    CONSTRAINT article_pkey PRIMARY KEY (pubid),
    CONSTRAINT article_pubkey_key UNIQUE (pubkey)
)

CREATE TABLE book
(
    pubid integer NOT NULL DEFAULT nextval('publication_pubid_seq '::regclass),
    pubkey text,
    title text,
    year integer,
    publisher text,
    isbn text,
    CONSTRAINT book_pkey PRIMARY KEY (pubid),
    CONSTRAINT book_pubkey_key UNIQUE (pubkey)
)

CREATE TABLE incollection
(
    pubid integer NOT NULL DEFAULT nextval('publication_pubid_seq '::regclass),
    pubkey text,
    title text,
    year integer,
    booktitle text,
    publisher text,
    isbn text,
    CONSTRAINT incollection_pkey PRIMARY KEY (pubid),
    CONSTRAINT incollection_pubkey_key UNIQUE (pubkey)
)

CREATE TABLE inproceedings
(
    pubid integer NOT NULL DEFAULT nextval('publication_pubid_seq '::regclass),
    pubkey text,
    title text,
    year integer,
    booktitle text,
    editor text,
    CONSTRAINT inproceedings_pkey PRIMARY KEY (pubid),
CONSTRAINT inproceedings_pubkey_key UNIQUE (pubkey)
)

CREATE TABLE author
(
    id serial NOT NULL,
    name text,
    homepage text,
    CONSTRAINT author_pkey PRIMARY KEY (id)
)

CREATE TABLE authored_article
(
    authorid integer NOT NULL,
    pubid integer NOT NULL,
    authornumber integer,
    CONSTRAINT authored_article_pkey PRIMARY KEY (authorid, pubid),
    CONSTRAINT authored_article_authorid_fkey FOREIGN KEY (authorid)
        REFERENCES author (id) MATCH SIMPLE
        ON UPDATE NO ACTION ON DELETE CASCADE,
    CONSTRAINT authored_article_pubid_fkey FOREIGN KEY (pubid)
        REFERENCES article (pubid) MATCH SIMPLE
        ON UPDATE NO ACTION ON DELETE CASCADE
)

CREATE TABLE authored_book
(
    authorid integer NOT NULL,
    pubid integer NOT NULL,
    authornumber integer,
    CONSTRAINT authored_book_pkey PRIMARY KEY (authorid, pubid),
    CONSTRAINT authored_book_authorid_fkey FOREIGN KEY (authorid)
        REFERENCES author (id) MATCH SIMPLE
        ON UPDATE NO ACTION ON DELETE CASCADE,
    CONSTRAINT authored_book_pubid_fkey FOREIGN KEY (pubid)
        REFERENCES book (pubid) MATCH SIMPLE
        ON UPDATE NO ACTION ON DELETE CASCADE
)

CREATE TABLE authored_incollection
(
    authorid integer NOT NULL,
    pubid integer NOT NULL,
    authornumber integer,
    CONSTRAINT authored_incollection_pkey PRIMARY KEY (authorid, pubid),
    CONSTRAINT authored_incollection_authorid_fkey FOREIGN KEY (authorid)
        REFERENCES author (id) MATCH SIMPLE
        ON UPDATE NO ACTION ON DELETE CASCADE,
    CONSTRAINT authored_incollection_pubid_fkey FOREIGN KEY (pubid)
        REFERENCES incollection (pubid) MATCH SIMPLE
        ON UPDATE NO ACTION ON DELETE CASCADE
)

CREATE TABLE authored_inproceedings
(
    authorid integer NOT NULL,
    pubid integer NOT NULL,
    authornumber integer,
    CONSTRAINT authored_inproceedings_pkey PRIMARY KEY (authorid, pubid),
    CONSTRAINT authored_inproceedings_authorid_fkey FOREIGN KEY (authorid)
        REFERENCES author (id) MATCH SIMPLE
        ON UPDATE NO ACTION ON DELETE CASCADE,
    CONSTRAINT authored_inproceedings_pubid_fkey FOREIGN KEY (pubid)
REFERENCES inproceedings (pubid) MATCH SIMPLE
ON UPDATE NO ACTION ON DELETE CASCADE
)

-- B-tree indices on referencing foreign key attributes to accelerate traditional joins
CREATE INDEX authored_article_authorid_idx
  ON authored_article
  USING btree
  (authorid);

CREATE INDEX authored_article_pubid_idx
  ON authored_article
  USING btree
  (pubid);

CREATE INDEX authored_book_authorid_idx
  ON authored_book
  USING btree
  (authorid);

CREATE INDEX authored_book_pubid_idx
  ON authored_book
  USING btree
  (pubid);

CREATE INDEX authored_incollection_authorid_idx
  ON authored_incollection
  USING btree
  (authorid);

CREATE INDEX authored_incollection_pubid_idx
  ON authored_incollection
  USING btree
  (pubid);

CREATE INDEX authored_inproceedings_authorid_idx
  ON authored_inproceedings
  USING btree
  (authorid);

CREATE INDEX authored_inproceedings_pubid_idx
  ON authored_inproceedings
  USING btree
  (pubid);
Appendix E

DBLP Queries

E.1 Freely Acyclic Conjunctive Queries

Original

```
SELECT name FROM author
WHERE EXISTS
  ( SELECT 1
    FROM authored_book
    WHERE authored_book.authorid = author.id
  )
```

```
SELECT name FROM author
WHERE EXISTS
  ( SELECT 1
    FROM structural_index.authored_book_authorid_fkey
    WHERE authored_book_authorid_fkey.to_id = author.block_id
  )
```

Query FACQ IP: select names of authors who wrote books

Original

```
SELECT name FROM author
WHERE EXISTS
  ( SELECT 1
    FROM authored_book
    WHERE authored_book.authorid = author.id
  )
AND EXISTS
  ( SELECT 1
    FROM authored_incollection
    WHERE authored_incollection.authorid = author.id
  )
```

```
SELECT name FROM author
WHERE EXISTS
  ( SELECT 1
    FROM structural_index.authored_book_authorid_fkey
    WHERE authored_book_authorid_fkey.to_id = author.block_id
  )
AND EXISTS
  ( SELECT 1
    FROM structural_index.authored_incollection_authorid_fkey
    WHERE authored_incollection_authorid_fkey.to_id = author.block_id
  )
```

Query FACQ IQ: select names of authors who wrote books and incollections
APPENDIX E. DBLP QUERIES

Original

```
SELECT name
FROM author
WHERE EXISTS
(  
  SELECT 1
  FROM authored_article
  WHERE authored_article.authorid = author.id
)
AND EXISTS
(  
  SELECT 1
  FROM authored_book
  WHERE authored_book.authorid = author.id
)
AND EXISTS
(  
  SELECT 1
  FROM authored_incollection
  WHERE authored_incollection.authorid = author.id
)
AND EXISTS
(  
  SELECT 1
  FROM authored_inproceedings
  WHERE authored_inproceedings.authorid = author.id
)
```

$k > 0$

```
SELECT name
FROM author
WHERE EXISTS
(  
  SELECT 1
  FROM structural_index.authored_article_authorid_fkey
  WHERE authored_article_authorid_fkey.to_id = author.block_id
)
AND EXISTS
(  
  SELECT 1
  FROM structural_index.authored_book_authorid_fkey
  WHERE authored_book_authorid_fkey.to_id = author.block_id
)
AND EXISTS
(  
  SELECT 1
  FROM structural_index.authored_incollection_authorid_fkey
  WHERE authored_incollection_authorid_fkey.to_id = author.block_id
)
AND EXISTS
(  
  SELECT 1
  FROM structural_index.authored_inproceedings_authorid_fkey
  WHERE authored_inproceedings_authorid_fkey.to_id = author.block_id
)
```

Query FACQ TR: select names of authors who wrote all publication types
Query FACQ 3P: select titles of articles from authors who also wrote books
**APPENDIX E. DBLP QUERIES**

<table>
<thead>
<tr>
<th>$k = 1$</th>
<th>$k = 2$</th>
<th>$k &gt; 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SELECT title</code>&lt;br&gt;<code>FROM book</code>&lt;br&gt;<code>WHERE EXISTS</code>&lt;br&gt;<code>( SELECT 1</code>&lt;br&gt;<code>  FROM authored_book</code>&lt;br&gt;<code>  WHERE authored_book.pubid = book.pubid</code>&lt;br&gt;<code>  AND EXISTS</code>&lt;br&gt;<code>    ( SELECT 1</code>&lt;br&gt;<code>      FROM authored_article</code>&lt;br&gt;<code>      WHERE authored_article.authorid = authored_book.authorid</code>&lt;br&gt;<code>    ) )</code>&lt;br&gt;<code>)</code></td>
<td><code>SELECT title</code>&lt;br&gt;<code>FROM book</code>&lt;br&gt;<code>WHERE EXISTS</code>&lt;br&gt;<code>( SELECT 1</code>&lt;br&gt;<code>  FROM authored_book</code>&lt;br&gt;<code>  WHERE authored_book.pubid = book.pubid</code>&lt;br&gt;<code>  AND EXISTS</code>&lt;br&gt;<code>    ( SELECT 1</code>&lt;br&gt;<code>      FROM author</code>&lt;br&gt;<code>      WHERE author.id = authored_book.authorid</code>&lt;br&gt;<code>      AND EXISTS</code>&lt;br&gt;<code>        ( SELECT 1</code>&lt;br&gt;<code>          FROM structural_index.authored_article_authorid_fkey</code>&lt;br&gt;<code>          WHERE authored_article_authorid_fkey.to_id = author.block_id</code>&lt;br&gt;<code>        ) )</code>&lt;br&gt;<code>)</code></td>
<td><code>SELECT title</code>&lt;br&gt;<code>FROM book</code>&lt;br&gt;<code>WHERE EXISTS</code>&lt;br&gt;<code>( SELECT 1</code>&lt;br&gt;<code>  FROM structural_index.authored_book_pubid_fkey</code>&lt;br&gt;<code>  WHERE authored_book_pubid_fkey.to_id = book.block_id</code>&lt;br&gt;<code>  AND EXISTS</code>&lt;br&gt;<code>    ( SELECT 1</code>&lt;br&gt;<code>      FROM structural_index.authored_book_authorid_fkey</code>&lt;br&gt;<code>      WHERE authored_book_authorid_fkey.from_id = authored_book_pubid_fkey.from_id</code>&lt;br&gt;<code>      AND EXISTS</code>&lt;br&gt;<code>        ( SELECT 1</code>&lt;br&gt;<code>          FROM structural_index.authored_article_authorid_fkey</code>&lt;br&gt;<code>          WHERE authored_article_authorid_fkey.to_id = authored_book_authorid_fkey.to_id</code>&lt;br&gt;<code>        ) )</code>&lt;br&gt;<code>)</code></td>
</tr>
</tbody>
</table>

Query FACQ 3R: select titles of books from authors who also wrote articles.
E.1. FREELY ACYCLIC CONJUNCTIVE QUERIES

SELECT title FROM book WHERE EXIST
( SELECT 1
  FROM authored_book
  WHERE authored_book.pubid = book.pubid AND EXIST
  ( SELECT 1
    FROM authored_article
    WHERE authored_article.authorid = authored_book.authorid
  ) ) )

Original

$k = 1$

SELECT title FROM book WHERE EXIST
( SELECT 1
  FROM authored_book
  WHERE authored_book.pubid = book.pubid
  AND EXIST
  ( SELECT 1
    FROM structural_index.authored_book_authorid_fkey
    WHERE authored_book_authorid_fkey.from_id = authored_book.block_id
  ) )

$k = 2$

SELECT title FROM book WHERE EXIST
( SELECT 1
  FROM structural_index.authored_book_pubid_fkey
  WHERE authored_book_pubid_fkey.to_id = book.block_id
  AND EXIST
  ( SELECT 1
    FROM structural_index.authored_book_authorid_fkey
    WHERE authored_book_authorid_fkey.from_id = authored_book_pubid_fkey.from_id
  ) )

$k > 2$

Query FACQ 3S: select titles of books from authors who also wrote articles and inproceedings
APPENDIX E. DBLP QUERIES

```
SELECT name FROM author WHERE EXISTS
    ( SELECT 1 FROM authored_article a1 WHERE a1.authorid = author.id AND EXISTS
        ( SELECT 1 FROM authored_article a2 WHERE a2.pubid = a1.pubid AND EXISTS
            ( SELECT 1 FROM authored_book WHERE authored_book.authorid = a2.authorid
                )
        )
    )

SELECT name FROM author WHERE EXISTS
    ( SELECT 1 FROM authored_article a1 WHERE a1.authorid = author.id AND EXISTS
        ( SELECT 1 FROM authored_article a2 WHERE a2.pubid = a1.pubid AND EXISTS
            ( SELECT 1 FROM author a3 WHERE a3.id = a2.authorid AND EXISTS
                ( SELECT 1 FROM structural_index.authored_book_authorid_fkey a4
                    WHERE a4.to_id = a3.block_id
                )
            )
        )
    )

SELECT name FROM author WHERE EXISTS
    ( SELECT 1 FROM authored_article a1 WHERE a1.authorid = author.id AND EXISTS
        ( SELECT 1 FROM authored_article a2 WHERE a2.pubid = a1.pubid AND EXISTS
            ( SELECT 1 FROM structural_index.authored_article_authorid_fkey a4
                WHERE a4.from_id = a2.block_id AND EXISTS
                ( SELECT 1 FROM structural_index.authored_article_authorid_fkey a5
                    WHERE a5.to_id = a4.to_id
                )
            )
        )
    )

SELECT name FROM author WHERE EXISTS
    ( SELECT 1 FROM authored_article a1 WHERE a1.authorid = author.id AND EXISTS
        ( SELECT 1 FROM structural_index.authored_article_authorid_fkey a1
            WHERE a1.to_id = author.block_id AND EXISTS
            ( SELECT 1 FROM structural_index.authored_article_pubid_fkey a2
                WHERE a2.from_id = a1.from_id AND EXISTS
                ( SELECT 1 FROM structural_index.authored_article_pubid_fkey a3
                    WHERE a3.to_id = a2.to_id AND EXISTS
                    ( SELECT 1 FROM structural_index.authored_article_authorid_fkey a4
                        WHERE a4.from_id = a3.from_id AND EXISTS
                        ( SELECT 1 FROM structural_index.authored_book_authorid_fkey a5
                            WHERE a5.to_id = a4.to_id
                        )
                    )
                )
            )
        )
    )

Query FACQ 5P: select names of authors from articles that have an author that wrote books
```
E.2 Guarded Fragment Queries

SELECT title
FROM article
WHERE NOT EXISTS
  ( SELECT 1
    FROM authored_article
    WHERE article.pubid = authored_article.pubid
  )

$\text{Query GF 1P: select titles of articles without author}$
APPENDIX E. DBLP QUERIES

SELECT name
FROM author
WHERE NOT EXISTS
  (SELECT 1
   FROM authored_article
   WHERE author.id = authored_article.authorid
  )
AND NOT EXISTS
  (SELECT 1
   FROM authored_book
   WHERE author.id = authored_book.authorid
  )
AND NOT EXISTS
  (SELECT 1
   FROM authored_incollection
   WHERE author.id = authored_incollection.authorid
  )
AND NOT EXISTS
  (SELECT 1
   FROM authored_inproceedings
   WHERE author.id = authored_inproceedings.authorid
  )

SELECT name
FROM author
WHERE block_id IN
  (SELECT id
   FROM structural_index.author index_author
   WHERE NOT EXISTS
     (SELECT 1
      FROM structural_index.authored_article_authorid_fkey
      WHERE authored_article_authorid_fkey.to_id = index_author.id
     )
   AND NOT EXISTS
     (SELECT 1
      FROM structural_index.authored_book_authorid_fkey
      WHERE authored_book_authorid_fkey.to_id = index_author.id
     )
   AND NOT EXISTS
     (SELECT 1
      FROM structural_index.authored_incollection_authorid_fkey
      WHERE authored_incollection_authorid_fkey.to_id = index_author.id
     )
   AND NOT EXISTS
     (SELECT 1
      FROM structural_index.authored_inproceedings_authorid_fkey
      WHERE authored_inproceedings_authorid_fkey.to_id = index_author.id
     )
  )

Query GF 1Q: select names of authors without publication
Original

```sql
SELECT name
FROM author
WHERE NOT EXISTS
  ( SELECT 1
    FROM authored_article
    WHERE author.id = authored_article.authorid
  )
AND EXISTS
  ( SELECT 1
    FROM authored_book
    WHERE author.id = authored_book.authorid
  )
AND EXISTS
  ( SELECT 1
    FROM authored_incollection
    WHERE author.id = authored_incollection.authorid
  )
AND EXISTS
  ( SELECT 1
    FROM authored_inproceedings
    WHERE author.id = authored_inproceedings.authorid
  )
```
APPENDIX E. DBLP QUERIES

Original

```sql
SELECT title FROM book
WHERE NOT EXISTS
  ( SELECT 1 FROM authored_book
    WHERE authored_book.pubid = book.pubid AND EXISTS
      ( SELECT 1 FROM authored_article
        WHERE authored_article.authorid = authored_book.authorid
      ) )
  AND EXISTS
  ( SELECT 1 FROM authored_book
    WHERE authored_book.pubid = book.pubid AND NOT EXISTS
      ( SELECT 1 FROM authored_article
        WHERE authored_article.authorid = authored_book.authorid
      ) )

k = 1

```sql
SELECT title FROM book
WHERE NOT EXISTS
  ( SELECT 1 FROM authored_book WHERE authored_book.pubid = book.pubid AND EXISTS
    ( SELECT 1 FROM structural_index.authored_book_authorid_fkey
      WHERE authored_book_authorid_fkey.to_id = author.block_id
    ) )
  AND EXISTS
  ( SELECT 1 FROM authored_book WHERE authored_book.pubid = book.pubid AND EXISTS
    ( SELECT 1 FROM structural_index.authored_book_authorid_fkey
      WHERE authored_book_authorid_fkey.to_id = authored_book_authorid_fkey.to_id
    ) )

k = 2

```sql
SELECT title FROM book WHERE NOT EXISTS
  ( SELECT 1 FROM structural_index.authored_book_pubid_fkey
    WHERE authored_book_pubid_fkey.to_id = book.block_id AND EXISTS
      ( SELECT 1 FROM structural_index.authored_book_authorid_fkey
        WHERE authored_book_authorid_fkey.from_id = authored_book.block_id AND EXISTS
          ( SELECT 1 FROM structural_index.authored_article_authorid_fkey
            WHERE authored_article_authorid_fkey.to_id = authored_book_authorid_fkey.to_id
          ) )
    )
  AND EXISTS
  ( SELECT 1 FROM structural_index.authored_book_pubid_fkey
    WHERE authored_book_pubid_fkey.to_id = book.block_id AND EXISTS
    ( SELECT 1 FROM structural_index.authored_book_authorid_fkey
      WHERE authored_book_authorid_fkey.from_id = authored_book.block_id AND NOT EXISTS
      ( SELECT 1 FROM structural_index.authored_article_authorid_fkey
        WHERE authored_article_authorid_fkey.to_id = authored_book_authorid_fkey.to_id
      ) )
    )

k > 2

```sql
SELECT title FROM book WHERE NOT EXISTS
  ( SELECT 1 FROM structural_index.authored_book_pubid_fkey
    WHERE authored_book_pubid_fkey.to_id = book.block_id AND EXISTS
      ( SELECT 1 FROM structural_index.authored_book_authorid_fkey
        WHERE authored_book_authorid_fkey.from_id = authored_book_pubid_fkey.from_id
        AND EXISTS
          ( SELECT 1 FROM structural_index.authored_article_authorid_fkey
            WHERE authored_article_authorid_fkey.to_id = authored_book_authorid_fkey.to_id
          )
        )
    )
  AND EXISTS
  ( SELECT 1 FROM structural_index.authored_book_pubid_fkey
    WHERE authored_book_pubid_fkey.to_id = book.block_id AND EXISTS
    ( SELECT 1 FROM structural_index.authored_book_authorid_fkey
      WHERE authored_book_authorid_fkey.from_id = authored_book_pubid_fkey.from_id AND NOT EXISTS
      ( SELECT 1 FROM structural_index.authored_article_authorid_fkey
        WHERE authored_article_authorid_fkey.to_id = authored_book_authorid_fkey.to_id
      )
    )
  )
```

Query GF 3P: select names of books that are only written by authors who never wrote articles
Query GF 3Q: select names of books that are only written by authors who also wrote articles

```sql

k = 1


k = 2


k > 2

```