Computing Covers of Query Results

Radu-Bogdan Berteanu

May 22, 2018

Abstract

Join queries are of crucial importance in database systems, and their efficient evaluation is a core task of query optimizers. Unfortunately, computing a join result is expensive and traditionally dominated by the output size, which hides redundant information in the form of Cartesian products. Our work will revolve around *Covers*, which are small subsets of the join result from which one can efficiently recover all its tuples. They exploit the power of factorisation in relational algebra and are a relational encoding of minimal edge covers of the hypergraph of the query result. Previous research showed that, once constructed, they can be used to achieve orders of magnitude speed-up in various tasks, such as learning regression models. It is therefore highly desirable to have efficient ways of constructing them and this thesis will introduce and analyse three algorithms that achieve this. For each one of them, we will present a high-level description, then delve deeper into their implementation and finally show benchmarking results. We hope that our work will showcase the immense potential of covers and draw attention to their vast applicability by proving that the hard step, their construction, can in fact be solved efficiently.

Acknowledgements

I am incredibly grateful to my supervisor, Prof. Dan Olteanu, for believing in me and for offering me guidance and the opportunity to join his weekly research meetings. Equally, I would like to express my great appreciation to Ahmet Kara, whose kindness and great help have been invaluable.

Contents

Chapter 1

Introduction

Efficient storage and processing of large amounts of data turned out to be one of most important challenges faced by institutions and companies during the beginning of the 21st century. Engineers worldwide are struggling to manage the increasing volume of information stored by their employers and academics are devoting more and more attention to this topic. Given the immense volume of data that has to be stored, a crucial problem is identifying ways to compress it and hence improve the storage efficiency. Most of the solutions employed so far have used general data compression methods that do not take advantage of the structure of the data. The few that do exploit it have achieved tremendous success, such as Google's F1 distributed database system [8].

Factorised databases [2] [4] are the foundation of a new kind of database management systems, employing the same, familiar model of relations at the logical layer while using compact, factorised representations at the physical layer. These representations can be exponentially more succinct than the relations they represent while allowing for optimal enumeration delay of the flat relation with only minimal pre-computation time. They exploit the structure of the relation to eliminate the redundancy that naturally occurs in the results of most queries made to such database systems.

Previous work in factorised databases relied on lossless representations of query results called f-representations and d-representations. Out of the two, the latter is the most efficient, since it reuses common internal structures instead of recomputing them everytime. The tremendous practicality of such concepts proved beyond doubt and numerous applications [5], [6], [7] have used them to achieve orders of magnitude improvements in performance. Still, their representation form made difficult their adoption as a data representation model by mainstream database systems that rely on relational storage [4]. This is because they are directed acyclic graphs that encode the query result using nodes that can be either data values or the relational operators Cartesian product and union.

Covers [4] are relational encodings of d-representations, bridging the practical gap between the usefulness of factorised representations and the conventional listing representation needed by existing systems. They achieve the exact same succinctness as d-representations and moreover, are subsets of the query result. Together with a (fractional hypertree) decomposition [16] of the query, the entire query result can be recovered with only linearithmic time pre-computation and constant enumeration delay. Fitting the flexibility and succinctness of factorised representations into existing relational database systems is highly desirable and hence covers deserve special attention as the key component in achieving this goal.

For a join query Q, database $\mathbf{D}$, and a decomposition $\mathcal{T}$ of Q with fractional hypertree width w, a cover over $\mathcal{T}$ has size $\mathcal{O}(|\mathbf{D}|^w)$, while the listing representation of the result can be as large as $\Omega(|\mathbf{D}|^{\rho^*})$, where ρ^* is the fractional edge cover number of Q [10]. The factor between the fractional hypertree width and the fractional edge cover number can be as large as the number of relation symbols in Q. Hence covers can be exponentially more succinct than the full join result, especially in the presence of join and multivalued dependencies in the input relations, which can be exploited to obtain the exponential succinctness gap.

Covers enjoy numerous other benefits. Fundamentally, they are a compact relation that represents the result of a join of multiple input relations, and hence they can be used to reduce the communication cost between servers in distributed database systems that can ship covers instead of the expensive intermediate query results [8]. Moreover, subsequent processing can benefit from receiving a cover as input, since they provide access locality for disk operations that can now read the tuples of the cover from the same disk page instead of reading tuples from different relations stored on different locations in memory. Furthermore, since covers are subsets of the query result, they can improve cache locality of these subsequent operations.

The close link between covers and d-represenstations is not just conceptual. A cover K can be translated into a d-representation of size $\mathcal{O}(|K|)$ and in time $\tilde{\mathcal{O}}(|K|)$, where we use $\tilde{\mathcal{O}}$ to hide a logarthmic factor. This immediately extends the applicability of covers to known workloads over factorised representations, such as in-database optimisation [11] [12] and in particular learning linear regression models [5]. These results are part of recent efforts to bring analytics inside the database [13] [14] in order to avoid the significant time spent on data import/export between database systems and statistical packages. In this thesis we will also show that we can transform a d-representation back into a cover. In fact, we will not even need to first materialise the factorised representation, but instead compute the cover directly over a d-tree.

We will start to introduce our contributions by first giving an example of a factorised representation of a join query result and a cover of the same join result.

1.1 An Example of Factorisation

To illustrate the power of factorised representations and covers of join results, consider the example given in Figure 1.1, where three relations, R_1, R_2, R_3 are joined. If we take a closer look at the natural join, we observe that it exhibits a high degree of redundancy. For example, the tuple $(c_2\ d_2)$ from R_3 appears four times in the result, and so does $(c_2\ d_2)$, and these two tuples take up 33% of the total size needed to represent the join result. Observe that this is the same amount of data needed to store the entire cover, which has only four tuples, the same as the input relations.

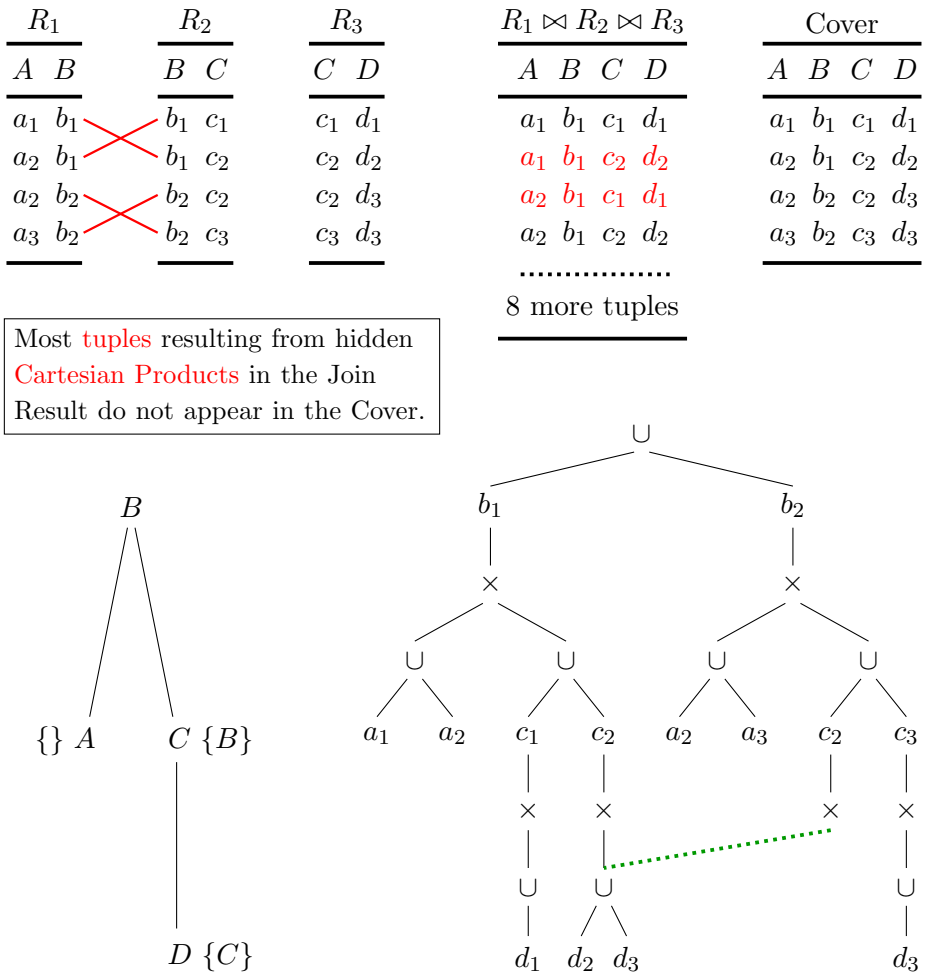

Figure 1.1: Top row: Relations R_1, R_2, R_3, the result of their natural join and a cover of this join result over the given d-tree; Bottom row: (left) a d-tree of the join query and (right) a d-representation of the join result over the same d-tree.

This redundancy is due mainly to the Cartesian products that are formed between subsets of the tuples in the input relations. The join result has to materialise them, whereas the cover and the factorised representation exploit their existence to compress this result. We can trace the compression process by inspecting the d-tree and the d-representation shown in the figure. The d-tree establishes an order among the attributes that will dictate the structure of the factorisation process. For each B value, the d-representation records those A values and C values that will (*independent* of each other) appear in the join result together with this particular B value. The key observation is that this information is stored only once, instead of repeating it for every corresponding D value, as happens in the join result. It is a simple application of the distributivity of the Cartesian product over union, but applied recursively down the tree leads to a (factorised) structure that succinctly encodes only the links between tuples in the input relation, yet can efficiently reconstruct the entire join result when needed.

Now let us take a closer look at the relation between the structure of the d-representation and the particular subset of tuples from the join result that form the cover. This relation is closely linked with the way the tuples of the join result are recovered from a factorised representation. To recompute the natural join of R_1, R_2, R_3 from the d-representation, one only needs to follow the structure of the tree, picking one branch at each union node and all branches at a product node, until all tuples in the join result have been recovered. To compute a cover, it suffices to pick only enough such paths to *cover* all the value nodes (e.x. a_1 or b_2, etc.) in the d-representation. That is, a cover will satisfy two properties:

Coverage Each value node in the d-representation appears in at least one tuple of the cover.

Minimality Every strict subset of a cover violates the above property (so there is no redundant information in a cover).

Note that the union of D values $\{d_2, d_3\}$ is shared by the branches (b_1, c_2) and (b_2, c_2): a d-representation achieves further succinctness compared to an f-representation by employing *caching*. This is achieved by noting that, if we are given a particular C-value, we can exactly identify the D-values that will appear in the join result in the same tuple with the C-value: all we have to do is inspect the relation R_3. Since attribute D does not appear in the same relation with any other attribute other than C, when identifying a union of D-values no additional information can be obtained by knowing the value of an attribute other than C. This is reflected in the fact that in the d-tree, the *key* of D is the attribute C. Hence, for a given C-value we can store the corresponding D-values only once and simply point to them when needed instead of recomputing them every time.

Now that we have explained the basics of factorised representations and covers, we are in a better position to describe our contributions.

1.2 Our Contributions

We have seen that factorised representations of a join result can achieve exponential succinctness compared to the flat representation. Moreover, covers, as a direct equivalent of d-representations, enjoy all the power and benefits of the latter while, crucially, being a relation themselves. The question that remains is thus how to efficiently construct a cover from the input relations and join query. Our thesis has three main contributions that aim to resolve this:

1. Chapter 3 presents an implementation of an algorithm based on Cover-Join Plans [4]. They represent a compositional approach that constructs covers in worst case optimal time and is well suited for implementation in existing database management systems. Their drawback is that for cyclic join queries they need to materialise intermediary join results prior to computing the cover. We will assess the degree to which this extra memory requirement affects the performance of the algorithm. Overall, this approach will represent the benchmark against which we will test our proposed algorithms.

2. In Chapter 4 we will introduce a novel algorithm, *F-Cover*, which aims to mitigate the need to pre-compute intermediary join results. It achieves this by covering the join result "one attribute at a time", according to the structure of an f-tree, a tree decomposition very similar to the already encountered d-tree (see Figure 1.1). We will see in Chapter 6 that this algorithm indeed performs very well for particular classes of queries, such as the hierarchical queries. Empirically we observe that it has high throughput, but can produce very large covers, and we will prove a tight upper bound on the sizes of such covers.

3. The last part of this thesis presents our flagship algorithm, the result of a sustained effort aimed at finding a method to efficiently build compact covers that does not need to pre-compute intermediary join results. It uses d-trees to construct covers that, compared to *F-Cover*, are up to a factor of $\mathcal{O}(|D|^{\log |S|})$ smaller, where S is the total number of attributes that appear in the join query. We observe empirically that it indeed produces the smallest covers while always being at most a constant factor away from the fastest algorithm.

Our thesis therefore has two main components, one of presenting and unifying theoretical results related to factorised databases and covers and a second part, which involves constructing, implementing and analysing algorithms that compute covers under stringent memory and time complexity constraints. Our work lies at the theoretical and practical foundation of a new kind of in-database computation that leverages the power and succinctness of covers to achieve orders of magnitute speed-ups.

Chapter 2

Preliminaries

We give here a set of definitions that apply to all chapters of this thesis. Further concepts will be introduced and defined in every chapter, as needed for the particular algorithm that is being presented.

2.1 Databases and Natural Join Queries

We consider relational databases with named attributes whose domain, $dom(\cdot)$, is over an ordered set of data values. A relation schema is a finite set of attributes. A tuple t over a relation schema S maps every attribute A in S to a value in $dom(A)$. A relation $\mathbf{R}$ over a schema S is a finite set of tuples over S. As a shorthand, we write $\mathbf{R}(S)$ to express that the schema of $\mathbf{R}$ is S. A database schema is a finite set of relation symbols. A database $\mathbf{D}$ over a database schema S contains a relation $\mathbf{R}$ for each relation symbol R in S. The size $|\mathbf{R}|$ of a relation $\mathbf{R}$ is the number of its tuples. The size $|\mathbf{D}|$ of a database $\mathbf{D}$ is the sum of the sizes of its relations.

We consider natural join queries of the form $Q = R_1(S_1) \bowtie \ldots \bowtie R_n(S_n)$ where each relation symbol is used at most once and the relation symbols refer to distinct relations. It can be shown that this is without loss of generality for both covers and factorised representations [4], [2]. The set $att(Q)$ of attributes of Q is the union of the schemas of its relation symbols. The *size* $|Q|$ of a query Q is the number of relation symbols that appear in Q. A database is said to be *globally consistent* [1] with respect to a query Q if no relation in the database contains a tuple that does not contribute to the query result. As a special case, two relations R_1 and R_2 are called *consistent* [4] if for every tuple t_1 of R_1 there is at least one tuple t_2 in R_2 such that t_1 and t_2 have the same values for the attributes in $S(R_1) \cap S(R_2)$ *and* vice-versa. A database $\mathbf{D}$ is *pairwise consistent* if any two relations in $\mathbf{D}$ are consistent. Using the above assumptions, from now on, to improve readability, we will not distinguish between a relation symbol and the corresponding relation.

2.2 Hypergraphs and Query Decompositions

A *(multi-)hypergraph* is a pair $H = (V, E)$, where V is a set of vertices and E is a (multi-)set of subsets of V, the hyperedges of H. The hypergraphs we will use will have no isolated vertices, that is, for every $v \in V$ there exists at least an edge $e \in E$ such that $v \in e$. A *fractional edge cover* of a hypergraph H is a function γ mapping each edge of H to a non-negative number such that $\sum_{e \ni v} \gamma(e) \geq 1$ for each vertex v of H. The $weight(\gamma)$ of the fractional edge cover γ is defined to be $\sum_{e \in E} \gamma(e)$. The *fractional edge cover number* $\rho^*(H)$ of H is defined as the minimum weight over all factional edge covers of H. We will view join queries as hypergraphs in the following, standard way:

Hypergraph of query The hypergraph H of a join query Q has one vertex $v = A$ for each attribute A in $att(Q)$ and one edge $e = S(R)$ for each relation R that appears in Q. Figure 2.1 shows an example.

A *hypertree decomposition* $\mathcal{T}$ of a hypergraph H is a pair $(T, \{B_t\}_{t \in T})$ where T is a tree and $\{B_t\}_{t \in T}$ is a family of subsets of vertices from H (called *bags*), one for each node in T [9]. A hypertree decomposition must satisfy two properties. *Edge coverage*: for each edge $e \in H$, there exists a bag B_t such that $e \subseteq B_t$. *Tree connectivity*: for each vertex v in H the set of nodes in T whose bags contain v, i.e. $\{t \in T | v \in B_t\}$, must induce a connected subtree in T. We define $B(\mathcal{T})$ to be the set of bags of $\mathcal{T}$: $B(\mathcal{T}) = \{B_t | t \in T\}$. The *fractional hypertree width* of $\mathcal{T}$ is defined to be $\max_{t \in T}\{\rho^*(B_t)\}$, that is, the maximum fractional edge cover number over all subgraphs obtained by restricting H to each bag in $B(\mathcal{T})$ [15].

The *fractional hypertree width* of a hypergraph H, denoted by fhtw($\mathcal{T}$), is the minimum possible (fractional hypertree) width of any hypertree decomposition of H [16]. We extend these notions to a query Q in the following natural way: a hypertree decomposition of Q and the fractional hypertree width of Q (denoted fhtw(Q)), are simply a hypertree decomposition and the fractional hypertree width of the hyperghraph of Q, respectively. We will refer to these concepts as simply a *decomposition* of Q and its *width*.

A join query Q is called *α-acyclic* [17] (or, simply, *acyclic*), if it has a decomposition in which each bag is contained in an edge of the hypergraph of Q. Notice that this type of decomposition has width one. We also define:

Hypergraph of query result For a join query Q, database $\mathbf{D}$, and tree decomposition $\mathcal{T}$ of Q, the hypergraph H of $Q(\mathbf{D})$ with respect to $\mathcal{T}$ has one vertex for each distinct tuple in $\pi_B Q(\mathbf{D})$ for each bag B in $B(\mathcal{T})$ and one edge for each tuple in $Q(\mathbf{D})$ [4]. Hence, the vertices of H are all the "mini" tuples obtained by projecting the query result $Q(\mathbf{D})$ on all bags B of $\mathcal{T}$. Moreover, the edge associated with a tuple t of $Q(\mathbf{D})$ covers all the "mini" tuples obtained when projecting t onto the bags of $\mathcal{T}$. Figure 2.1 shows an example of such a hypergraph.

2.3 Covers

Given a hypergraph H and a subset M of its edges, we say that M is an *edge cover* of H if each vertex in H is contained in at least one edge in M. Moreover, we say that M is a *minimal* edge cover if it is an edge cover and none of its strict subsets is. Notice that for a query Q with decomposition $\mathcal{T}$ and database $\mathbf{D}$, an edge cover M of the hypergraph of $Q(\mathbf{D})$ with respect to $\mathcal{T}$ corresponds to a subset of the tuples in $Q(\mathbf{D})$ (hence, a relation), since, by construction, there is a one to one mapping between tuples in $Q(\mathbf{D})$ and edges in its hypergraph. We say that such a relation is *induced* by M. We can then define covers [4] in terms of the hypergraph of the query result:

Definition 1 (Covers)**.** Let Q be a natural join query over database $\mathbf{D}$ and $\mathcal{T}$ a decomposition of Q. A relation K is a *cover* of the query result $Q(\mathbf{D})$ over $\mathcal{T}$ if it is induced by a minimal edge cover of the hypergraph of $Q(\mathbf{D})$ with respect to $\mathcal{T}$.

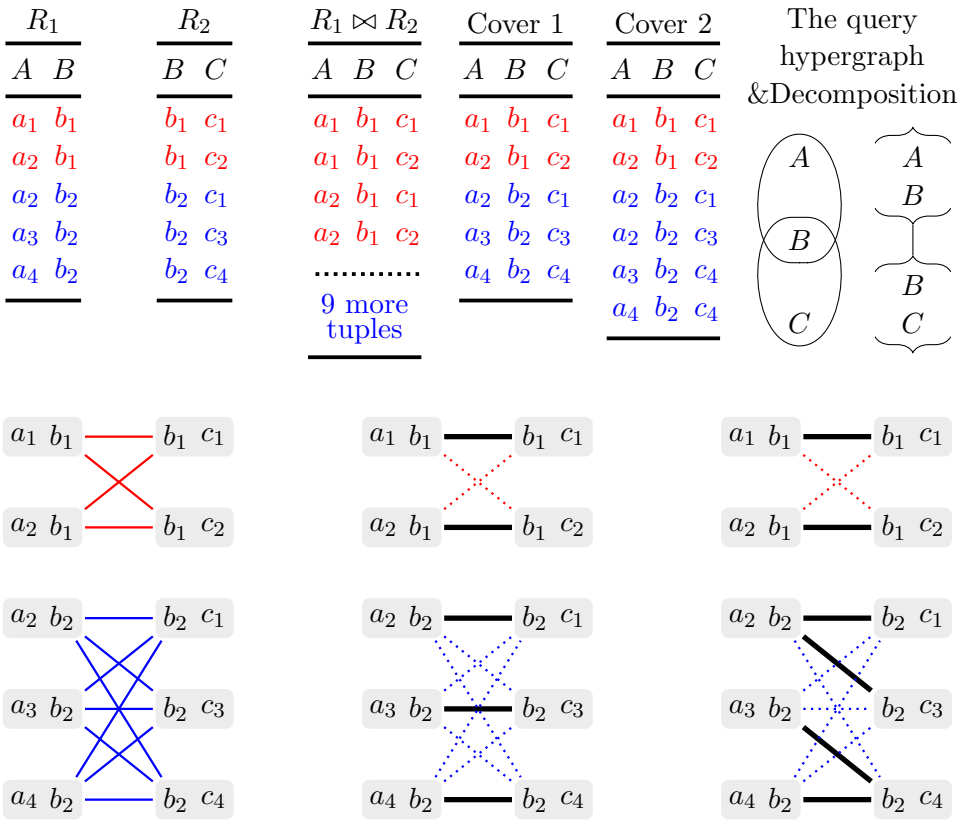

Figure 2.1: Bottom row: (left) (Hyper)graph H of the query result containing two complete bipartite subgraphs; (centre) A minimum edge cover of H that induces Cover 1; (right) A minimal edge cover that induces Cover 2.

Figure 2.1 gives an example of two covers of the result of an acyclic join query $R_1 \bowtie R_2$. Cover 1 is a minimum-size cover of this result, while Cover 2 is a maximum-size cover. Let us raise a couple of important points:

First, notice that a hypergraph H may admit multiple *minimal* edge covers. Among them, the ones with minimum size (number of edges) are called *minimum* edge covers and their size is known in the literature as the *edge cover number* of H. The definition of covers of query results is based on minimal edge covers, which avoids the intractability [18] of finding a minimum edge cover. Hence, different covers of a join result may have different sizes. Nevertheless, [4] shows that the maximum size of a cover K of the result of a join query Q over database D is $\mathcal{O}(|D|^{fhtw(Q)})$.

Secondly, for joins of exactly two relations, the hypergraph of the query result is a collection of disjoint complete bipartite subgraphs (here only two, colored red and blue), and hence we can reduce the computation of a cover of the join result to computing (disjoint) edge covers of the aforementioned disjoint subgraphs. This observation underpins the computation of the Cover-Join operator, which will be presented in Chapter 3.

We now present two propositions which, taken together, form an alternative definition of covers [4]. Let Q be a join query, $\mathcal{T}$ a decomposition of Q, $\mathbf{D}$ a database and K a cover of $Q(\mathbf{D})$ over $\mathcal{T}$. Then:

Proposition 2 (Result-preservation)**.** *Given* $(Q, \mathcal{T}, \mathbf{D})$*, the projection of* K *onto any of the bags in* $B(\mathcal{T})$ *equals the projection of* $Q(\mathbf{D})$ *onto the same bag, i.e.:* $\pi_B K = \pi_B Q(\mathbf{D})$*, for all bags* $B \in B(\mathcal{T})$*. We say that the cover* K *is* result-preserving*.*

Proposition 3 (Minimality)**.** *Given* $(Q, \mathcal{T}, \mathbf{D})$*, no strict subset of* K *is result-preserving. We say that* K *is* minimal*.*

Notice that Proposition 2 is a direct consequence of the fact that covers are induced by *edge covers* of the hypegraph of the query result, since the vertices of the latter are exactly the set $\bigcup_{B \in B(\mathcal{T})} \pi_B Q(\mathbf{D})$, and all vertices must be covered by at least one edge. In a similar manner, Proposition 3 follows from the fact that covers are induced by *minimal* edge covers of the query result. Thus these propositions can be considered properties of covers. Nevertheless, together, they form an alternative definition of covers and moreover, they are of high practical interest, since given a relation K we can check if it is a cover by checking if it is *minimal result-preserving*. For example, both Cover 1 (K_1) and Cover 2 (K_2) from Figure 2.1 are result-preserving, since taking their projections onto the bags $B_1 = \{A, B\}$ and $B_2 = \{B, C\}$ yields: $\pi_{B_1} K_1 = \pi_{B_1} K_2 = \pi_{B_1} Q(\mathbf{D}) = R_1$ and, moreover, $\pi_{B_2} K_1 = \pi_{B_2} K_2 = \pi_{B_2} Q(\mathbf{D}) = R_2$. Similarly, one can check that no strict subset of K_1 and K_2 is result-preserving. Lastly, notice that result preservation implies the following important property [4]:

Proposition 4 (Result recoverability)**.** $\bowtie_{B \in B(\mathcal{T})} \pi_B K = Q(\mathbf{D})$

Chapter 3

Computing Covers using Cover-Join Plans

We now begin the first chapter of a series of three, each of which will present a different method of computing covers. In this particular chapter we will present a method proposed in [4] which is based on two main concepts: the *Cover-Join Operator*, alluded to in Section 2.3, and *Cover-Join Plans*, which use the aforementioned operator in a recursive manner to compute covers of the result of a join of an arbitrary number of relations. This represents a *compositional* approach to computing covers and is different in style to the algorithms presented in later chapters.

This solution relies on first transforming the arbitrary input query and database into an *acyclic* query over a *globally consistent* database (Proposition 3 in [4]). More specifically, given a natural join query Q, decomposition $\mathcal{T}$ of Q and a database $\mathbf{D}$, by materialising the bags in $\mathcal{T}$ we obtain a triple $(Q', \mathcal{T}, \mathbf{D}')$ such that Q' is an acyclic natural join query, $\mathcal{T}$ a decomposition of Q', and $\mathbf{D}'$ is now a globally consistent database, comprised of exactly one relation per bag of $\mathcal{T}$. Below, we illustrate this for the bow-tie query:

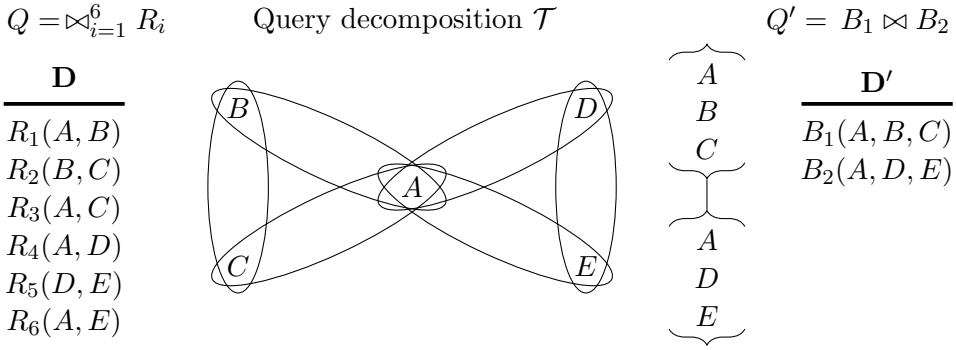

Figure 3.1: Transforming $(Q, \mathcal{T}, \mathbf{D})$ into $(Q', \mathcal{T}, \mathbf{D}')$ with Q' acyclic.

Notice that in Figure 3.1, the width of the decomposition $\mathcal{T}$ is 3/2 with respect to Q, but the same decomposition has width 1 with respect to Q' (Q' is acyclic). The reason why we need to transform the query into an acyclic one is that the Cover-Join Plans assume that there is a one-to-one mapping between the bags of the query decomposition and the relations in the database they operate over, which is only the case if the query is acyclic [17]. Moreover, they assume that the database is globally consistent, and so we will have to ensure that this holds for the new databse $\mathbf{D}'$.

This transformation has a non-trivial cost. The size of the new database $\mathbf{D}'$ is $\mathcal{O}(|\mathbf{D}|^{fhtw(\mathcal{T})})$. Nevertheless, it can be computed in time $\tilde{\mathcal{O}}(|\mathbf{D}|^{fhtw(\mathcal{T})})$, and this is asymptotically optimal with respect to the upper bound on the size of covers. An important aspect will then be assessing the actual performance impact this transformation has on the *Plan-Cover* algorithm.

3.1 The Cover-Join Operator

The cover-join operator [4] is a binary operator that constructs the cover of the result of a join of two relations. We remarked in Section 2.3 that the join of exactly two (consistent) relations is a collection of disjoint complete bipartite subgraphs (one subgraph for each common value of the join attributes of the two relations). Computing a cover of the result thus reduces to computing (disjoint) covers of the (sub-)relations induced by the subgraphs and combining the results. Now, the key observation underlying the efficiency of the cover-join operator is that computing an edge cover of a complete bipartite graph is easy, as we can simply pair the vertices of the two sides one by one until we exhaust the vertices on one of the two sides (reminiscent of the *zip* operation from functional programming), and then combine all the remaining vertices with any vertex from the exhausted side. By doing this for every complete bipartite subgraph and taking a union of the resulting tuples, the cover-join operator constructs a cover of minimum size of the result of the join of R_1 and R_2. Hence we define the following [4]:

Definition 5. Let $\mathbf{D} = \{R_1, R_2\}$ be (globally) consistent, $Q = R_1 \bowtie R_2$ a natural join query and $\mathcal{T}$ a decomposition of Q with $B(\mathcal{T}) = \{S(R_1), S(R_2)\}$. The cover-join of R_1 and R_2, denoted by $\overset{\circ}{\bowtie}$, computes a cover of $Q(\mathbf{D})$ of *minimum* size.

The size of the cover constructed is equal to the edge cover number of the hypergaph of the query result [4]. Notice that this hypergraph has exactly $|R_1| + |R_2|$ vertices, and hence this represents an upper bound on the size of the cover. The time complexity of the cover-join operator is dominated by the need to intersect the two relations on the common attributes (i.e. attributes in $C = S(R_1) \cap S(R_2)$). This can be implemented by sorting the relations (in parallel) on these attributes, leading to a $\mathcal{O}(|C| \cdot s \cdot \log s)$ running time, where $s = max(|R_1|, |R_2|)$.

3.2 Cover-Join Plans

Cover-Join Plans [4] compute a cover of the result of a join of an arbitrary number of relations by recursively splitting the (hypertree) decomposition $\mathcal{T}$ of an acyclic join query Q into (any) two non-empty sub-decompositions $\mathcal{T}_1$ and $\mathcal{T}_2$ that are connected by a single edge in $\mathcal{T}$ and then applying the binary cover-join operator to the two (sub-)covers computed over them:

Definition 6. For a natural acyclic join query Q, decomposition $\mathcal{T}$ of Q and globally consistent database $\mathbf{D}$, a *cover-join plan* $\varphi(\mathcal{T})$ computes a cover of $Q(\mathbf{D})$ over $\mathcal{T}$ of size $O(|\mathbf{D}|)$ and in time $\tilde{O}(|\mathbf{D}|)$, recursively, as follows:

- If $\mathcal{T}$ consists of one node R, then $\varphi(\mathcal{T}) = R$
- If $\mathcal{T}$ consists of more than two nodes, then split $\mathcal{T}$ into two non-empty subtrees $\mathcal{T}_1$ and $\mathcal{T}_2$ that are conntected by an edge. Then: $\varphi(\mathcal{T}) = \varphi(\mathcal{T}_1) \mathbin{\mathring{\bowtie}} \varphi(\mathcal{T}_2)$

Let us briefly present an example (in the spirit of the one found in [4]) that shows that a cover-join plan can construct covers of non-minimum size, depending on the exact output of each cover-join operator, which is only binary (hence *locally*) optimal. This is an important remark, since this limitation will extend to our implementation. Consider the join of three relations, R_1, R_2 and R_3 with the query decomposition $\mathcal{T}$ with bags: $B(\mathcal{T}) = \{\{A, B\}, \{B, C\}, \{C, D\}\}$, the Cover-Join Plan $\varphi = (R_1 \mathbin{\mathring{\bowtie}} R_2) \mathbin{\mathring{\bowtie}} R_3$ and two covers K_1 and K_2 that can be produced by the cover-join $R_1 \mathbin{\mathring{\bowtie}} R_2$:

R_1	
A	B
a_1	b_1
a_2	b_1
a_3	b_1

R_2	
B	C
b_1	c_1
b_1	c_2

R_3	
C	D
c_1	d_1
c_1	d_2
c_2	d_3

K_1		
A	B	C
a_1	b_1	c_1
a_2	b_1	c_2
a_3	b_1	c_2

K_2		
A	B	C
a_1	b_1	c_1
a_2	b_1	c_1
a_3	b_1	c_2

Now, if $R_1 \mathbin{\mathring{\bowtie}} R_2$ outputs K_1, then this will lead to a non-minimum cover for $R_1 \bowtie R_2 \bowtie R_3$, while K_2 leads to a mninimum cover of size 3:

$K_1 \mathbin{\mathring{\bowtie}} R_3$			
A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_1	c_1	d_2
a_2	b_1	c_2	d_3
a_3	b_1	c_2	d_3

$K_2 \mathbin{\mathring{\bowtie}} R_3$			
A	B	C	D
a_1	b_1	c_1	d_1
a_2	b_1	c_1	d_2
a_3	b_1	c_2	d_3

3.3 The *Plan-Cover* Algorithm

Now that we have presented all the preliminaries required for this section, let us introduce the Plan-Cover Algorithm, which represents an algorithmic description of the method of computing covers described in [4] and explained in the previous sections of this chapter. We will give a pseudocode of the algorithm, explain the implementation decisions and then benchmark it.

3.3.1 Pseudocode

Input:

- Q, an arbitrary natural join query
- $\mathcal{T}$, a decomposition of Q
- $\mathbf{D}$, an arbitrary database

Returns:

- a cover K of $Q(\mathbf{D})$

1. Compute and materialise the bags:
foreach node t of $\mathcal{T}$ **do**:
$P \leftarrow B_t$ // the attributes in the bag of t
$R_1^P, R_2^P, \ldots R_k^P \leftarrow \{\pi_P R | R \in S(\mathbf{D}), S(R) \cap P \neq \emptyset\}$
$R_{B_t} \leftarrow R_1^P \bowtie R_2^P \bowtie \ldots \bowtie R_k^P$
$\mathbf{D}' \leftarrow \mathbf{D}' \cup R_{B_t}$

2. Make $\mathbf{D}'$ globally consistent, in two phases:
foreach node t **in** $postOrderTraversal(\mathcal{T})$ **do**:
if t is not root **then**:
$R_{B_t} \ltimes R_{B_{parent(t)}}$

foreach node t **in** $preOrderTraversal(\mathcal{T})$ **do**:
if t is not root **then**:
$R_{B_{parent(t)}} \ltimes R_{B_t}$

3. Compute a cover of $Q(\mathbf{D}')$ with a cover-join plan:
$K \leftarrow foldl1\ (\mathring{\bowtie})\ (map\ getBag\ (preOrderTraversal(\mathcal{T}))$

return K

Figure 3.2: Pseudocode of the *Plan-Cover* Algorithm

As suggested in the pseudocode, the algorithm proceeds in three stages. Below we give details about each stage and its associated running time complexity, followed by formal proofs of correctness of the constructions on the next page.

1. **Materialising the bags**:
   For each node t in $\mathcal{T}$ we find all relations $R_1, R_2, \ldots R_k$ whose schema contains at least one attribute from the bags of t, i.e. B_t. We then use a *worst-case optimal* join algorithm to compute R_{B_t}, the natural join of these relations. Lastly, we add this join to the new database $\mathbf{D}'$ and also, implicitly, associate it to t. As described at the beginning of this chapter, the size of the new database $\mathbf{D}'$ is $\mathcal{O}(|\mathbf{D}|^{fhtw(\mathcal{T})})$ and the total running time of this stage of the algorithm is $\mathcal{O}(|\mathbf{D}'| \cdot \log(|\mathbf{D}'|))$

2. **Making the new database globally consistent**:
   We make $\mathbf{D}'$ globally consistent in two phases: a bottom-up pass through the tree $\mathcal{T}$, followed by a top-down pass. During the bottom-up pass, for each node t of $\mathcal{T}$, we remove from the bag of t's parent all tuples that do not have a corresponding tuple in t's bag, i.e. performing a semi-join. Then, during the top-down pass, for each node t of $\mathcal{T}$, we remove from t's bag all tuples that do not have a matching tuple in the bag of t's parent. To analyze the running time of a semi-join, let s be the sum of the sizes of two input bags, then a semi-join takes $\mathcal{O}(s \cdot \log(s))$ time, since we can perform a semi-join by sorting the two relations on their common (i.e. join) attributes, and discarding all tuples that do not have a corresponding tuple in the other relation. Therefore the total time required by both passes, and hence stage 2 of the algorithm is $\mathcal{O}(|\mathbf{D}'| \cdot \log(|\mathbf{D}'|))$.

3. **Computing a cover of the join result using a cover-join plan**:
   The last stage of the algorithm deserves particular attention. Notice that in the (recursive) definition of cover-join plans, the choice of edge at which to split the tree decomposition $\mathcal{T}$ is not specified. Since the upper bound on the total running time is the same ($\mathcal{O}(|\mathbf{D}'| \cdot \log(|\mathbf{D}'|))$) no matter what edge is chosen, the implementer is given the freedom to choose according to their own rules. We have decided to always split the tree at one of its leaves (in particular, the rightmost leaf), and we believe an elegant way to specify this is using the Haskell functions $foldl1$ and *map*. $foldl1$ is the variant of $foldl$ that takes as input a non-empty list (since $\mathcal{T}$ will have at least one node). We can think of the tree $\mathcal{T}$ as having an associated tree of bags, constructed by mapping each node t in $\mathcal{T}$ to its (materialised) bag, B_t. The binary cover-join operator is then applied recursively to reduce this tree of bags to a cover of the entire join result.

Let us now delve deeper into the first two stages of *Plan-Cover* and prove that the algorithm correctly implements the specification of these steps, including satisfying the running time constraints:

1. We have alluded to the use of a worst-case optimal join algorithm to compute the bags. This is in fact a very exciting, recent development in database theory and practice [19] [20]. Until recently, highly optimised systems, built over decades, have been using worst-case sub-optimal join algorithms, that is, algorithms that cannot compute a query result $Q(\mathbf{D})$ in $\mathcal{O}(|Q(\mathbf{D})|)$ time because the size of the intermediate results they compute can be, in the worst-case, bigger than the size of the final join result. Leapfrog Triejoin [19] is part of a new class of algorithms that run in linear time in the size of the join result (its standard implementation, described in their paper, runs in ime linearithmic in the size of the join result, and so will our implementation). Additionally, it has been shown that for a query Q and database $\mathbf{D}$, the size of the query result $Q(\mathbf{D})$ is $O(|\mathbf{D}|^{\rho^*})$, where ρ^* is the fractional edge cover number of the hypergraph of Q [16]. Now, consider the join query Q and decomposiiton $\mathcal{T}$ received as input by *Plan-Cover*, and let H be the hypergraph of Q. We have defined $fhtw(\mathcal{T})$ to be the maximum fractional edge cover number over all subgraphs obtained by restricting H to each of the bags of $\mathcal{T}$. Therefore, using Leapfrog Triejoin, we can compute each bag B_t in time $\tilde{O}(|B_t|) = \tilde{O}(|\mathbf{D}|^{fhtw(\mathcal{T})})$. Now, under data complexity, we consider the size of the query (and hence the number of bags) to be a constant, and therefore the total runing time required to materialise all bags is $\tilde{O}(|\mathbf{D}|^{fhtw(\mathcal{T})})$

2. In the literature, the two phasses, taken together, represent a *semi-join programme*. Their application makes the bags *pairwise* consistent [21]. For acyclic queries, pairwise consistency implies global consistency [1]. While a formal proof can be found in Theorem 6.4.5 of [1], we present here an intuitive explanation, partly based on [21]. Since $\mathcal{T}$ is a decomposition, for any attribute A in $att(Q)$, the nodes of $\mathcal{T}$ whose bag contains A induce a connected subtree of $\mathcal{T}$. Now, pick any node t of $\mathcal{T}$ and let S be any subset of attributes of its bag. Crucially, observe that the nodes of $\mathcal{T}$ whose bag *includes* S also forms a connected subtree $\mathcal{T}_S$ of $\mathcal{T}$. At the end of the bottom-up phase, the (materialised) bag $R_{B_{\text{top}}}$ of the topmost node in $\mathcal{T}_S$ will satisfy: $\pi_S R_{B_{\text{top}}} \subseteq \pi_S R_{B_t}$ for all t in $\mathcal{T}_S$. Immediately after the top-down phase semijoins $R_{B_{\text{top}}}$ with its parent, some tuples from $R_{B_{\text{top}}}$ may be discarded, but nevertheless, at the end of the top-down phase, *all* notes t in $\mathcal{T}_S$ will satisfy $\pi_S R_{B_t} \subseteq \pi_S R_{B_{\text{top}}}$, and hence $\pi_S R_{B_t} = \pi_S R_{B_{\text{top}}}$. This means that after the two phases, the new database will be pairwise consistent, because for any two materialised bags R' and R'' we can let $S = S(R') \cap S(R'')$ and obtain $\pi_S R' = \pi_S R''$, which implies that R' and R'' are now consistent.

3.3.2 Design Decisions During Implementation

A lot of thought and care went into implementing this algorithm in an efficient, extensible and flexible manner, following software engineering best practices. We have built the entire system, from the ground up, in Java 10, using a functional programming paradigm, making extensive use of the newly introduced Streams library. To illustrate, consider the code below, which is the main section of the *Engine* class and implements the pseudocode of Plan-Cover (Figure 3.2).

```
// 1. Compute and materialise the bags:
root.preOrderTraversal()
    .forEach(treeNode -> treeNode.materialiseBag(database));

// 2. Make D' globally consistent, in two phases:
root.postOrderTraversal()
    .filter(treeNode -> !treeNode.isRoot())
    .forEachOrdered(TreeNode::parentLeftSemiJoinThis);
root.preOrderTraversal()
    .filter(treeNode -> !treeNode.isRoot())
    .forEachOrdered(TreeNode::thisLeftSemiJoinParent);

// 3. Compute a cover of Q(D') with a cover-join plan:
cover = root.preOrderTraversal()
    .sequential()
    .map(TreeNode::getBag)
    .reduce(JoinAlgorithms::binaryCoverJoin)
    .get();
```

The project is modular, and among its components, four classes stand out. Together, they provide the core functionality of the system:

- *Engine*: Responsible for running the entire system. Corresposnds to the Plan-Cover pseudocode and its contents are listed above.
- *TreeNode*: Corresponds to a node of the decomposition of $\mathcal{T}$. It provides views in the form of pre- and post- order traversals and methods for semi-joining the node with its parent or children
- *JoinAlgorithms*: A utility library that holds all the main algorithms for joining relations (Leapfrog Triejoin implementation) and computing a binary cover-join
- *RelationInstance* A specialised implementation of the *Relation* abstract class that assumes that its data will not change, which allows for the efficient implementation of a Triejoin Iterator over it.

Chapter 4

Computing Covers over F-trees

In this chapter we present a monolithic approach to constructing covers, as opposed to the compositional one described in the previous sections. We will first introduce the concepts of f-representation and f-trees [2] which are the backbone of factorised databases. Using relational algebra, they decompose a flat join result into a tree, encoding the dependency between the values of the attributes that appear in the join. This decomposition follows a factorisation pattern in the form of an f-tree, which imposes a partial order over these attributes. We will see that an f-tree is a restricted type of hypertree decomposition of the query, and hence it cannot always lead to the same succinctness of the resulting cover as was possible when using cover-join plans. Nevertheless, this monolothic approach eliminates the need to compute intermediary join results, as we had to do when materialising the bags of the query decomposition. It will be interesting to see what effect this memory-time tradeoff has on our ability to efficiently compute covers.

Factorised representations of join results hence suggest a different way of approaching the problem of constructing covers. Instead of repeatedly computing the cover of the result of the join of two relations, as in the case of the cover-join plans, we will take a more global view of the query and try to cover the entire join result, one attribute at a time. We will build an algorithm that, guided by the structure of an f-tree, "covers" all the values in an f-representation, and will prove that this leads to a cover of the join result. Moreover, our algorithm will construct the cover without actually storing the f-representation, which can lead to dramatic memory savings. Only the *trace* of the computation will represent an f-representation. Moreover, in order to achieve wort-case optimality, our algorithm will leverage the subroutine used by Leapfrog Triejoin [19] to intersect multiple relations on a common attribute. The computation of the algorithm will hence be based on a generalisation of the Leapfrog-Triejoin algorithm [3].

4.1 F-representations and F-trees

We have seen an example of a factorised representation of a query result in Figure 1.1, although there we have depicted a d-representation, which differs from an f-representation only in the fact that is uses caching. Formally, f-representations are relational algebra expressions which represent relations using: Cartesian products (symbolising independence of attributes and data), union (representing alternative values of an attribute) and singletons, which are the atomic constructs used to represent values of attributes.

Definition 7 (F-representations [2])**.** An *f-representation* F over a non-empty schema S is a relational algebra expression that represents a relation $[\![F]\!]$. We define both F and $[\![F]\!]$ inductively, as follows:

- $[\![\emptyset]\!]$ is the empty relation over schema S;
- $[\![a]\!]$ is the relation $\{(a)\}$, i.e. a relation having one tuple with one data value, if $S = \{A\}$ and $a \in \mathrm{Dom}(A)$. We will call a a *singleton*;
- $[\![(F_1 \cup F_2 \cup \ldots \cup F_n)]\!] = [\![F_1]\!] \cup [\![F_2]\!] \cup \ldots \cup [\![F_n]\!]$, where each F_i is an f-representation over S;
- $[\![(F_1 \times F_2 \times \ldots \times F_n)]\!] = [\![F_1]\!] \times [\![F_2]\!] \times \ldots \times [\![F_n]\!]$, where each F_i is an f-representation over S_i such that $\{S_1, S_2, \ldots, S_n\}$ is a partition of S.

Next we define f-trees, which impose a structure over an f-representation and hence dictate the factorisation pattern of a query result:

Definition 8 (F-trees [2])**.** Given a natural join query Q, an *f-tree* $\mathcal{T}^f$ of Q is a rooted forest, having one node for each attribute in $att(Q)$, such that the *path condition* is satisfied: for each relation symbol R in Q, the attributes in $S(R)$ lie along the same root-to-leaf path in $\mathcal{T}^f$.

For a node $\mathcal{A}$ of an f-tree $\mathcal{T}^f$, we define $\mathcal{T}^f_{\mathcal{A}}$ to be the subtree of $\mathcal{T}^f$ rooted at $\mathcal{A}$ and define $anc(\mathcal{A})$ to be the set of all attributes that are ancestors of $\mathcal{A}$ in $\mathcal{T}^f$. We overload $anc(\cdot)$ to subtrees and forests in the natural way. Informally (for now), we say that f-representations are over f-trees if they follow the structure of the f-tree. To improve readability, syntactically, we will not differentiate between a node and its attribute.

The succinctness of f-representations over f-trees $\mathcal{T}^f$ is achieved by branching (i.e. nodes in $\mathcal{T}^f$ having multiple children), which exploits the *conditional independence* of attributes. Formally, for a node $\mathcal{A}$ of $\mathcal{T}^f$ having children $\mathcal{C}_1, \mathcal{C}_2, \ldots \mathcal{C}_n$, given the values for $anc(\mathcal{A})$, the values for attributes in $\mathcal{T}^f_{\mathcal{C}_i}$ are independent of the values for attributes in $\mathcal{T}^f_{\mathcal{C}_j}$, for any $i \neq j$. This allows us to factorise the f-representation over the subtree rooted at $\mathcal{A}$ into a product of f-representations over the subtrees rooted at its children.

Let us now use these notions to give a constructive definition of the f-representation of a query result $Q(\mathbf{D})$ over an f-tree $\mathcal{T}^f$, denoted $\mathcal{T}^f(Q(\mathbf{D}))$. This definition will form the basis of our algorithm for constructing covers of join results over f-trees. We will achieve this by inductively constructing an f-representation F *over* a subtree or forest $\mathcal{T}^f_\mathcal{X}$ of $\mathcal{T}^f$, *conditioned on* an assignment (tuple) t for the variables in $anc(\mathcal{X})$ and denoted $F(\mathcal{T}^f_\mathcal{X}, t)$. Intuitively, $F(\mathcal{T}^f_\mathcal{X}, t)$ represents the relation $\pi_\mathcal{X}\sigma_{anc(\mathcal{X})=t}Q(\mathbf{D})$.

Definition 9. For a natural join query Q, database $\mathbf{D}$, f-tree $\mathcal{T}^f$ of Q, we denote the f-representation of $Q(\mathbf{D})$ over $\mathcal{T}^f$ as $\mathcal{T}^f(Q(\mathbf{D}))$ and define it as $F(\mathcal{T}^f, \langle\rangle)$ where for any subtree or forest $\mathcal{X}$ in $\mathcal{T}^f$, and any tuple t over $anc(\mathcal{X})$, the f-representation $F(\mathcal{X}, t)$ is defined inductively as follows:

- If $\mathcal{X}$ is a leaf $\mathcal{A}$, then:

$$F(\mathcal{A}, t) = \bigcup_{a \in I} a, \quad \text{where } I = \bigcap_{\substack{R \in S(\mathbf{D}): \\ \mathcal{A} \in S(R)}} \pi_\mathcal{A}\sigma_{anc(\mathcal{A})=t}R$$

- If $\mathcal{X}$ is a subtree $\mathcal{T}^f_\mathcal{A}$ with root $\mathcal{A}$ and children $\mathcal{T}^f_1, \mathcal{T}^f_2, \ldots, \mathcal{T}^f_n$, then:

$$F\left(\mathcal{T}^f_\mathcal{A}, t\right) = \bigcup_{a \in I} \left(a \times F\left(\left\{\mathcal{T}^f_1, \mathcal{T}^f_2, \ldots, \mathcal{T}^f_n\right\}, t \times a\right)\right)$$

  where $I = \bigcap_{\substack{R \in S(\mathbf{D}): \\ \mathcal{A} \in S(R)}} \pi_\mathcal{A}\sigma_{anc(\mathcal{A})=t}R$.

- If $\mathcal{X}$ is a non-empty forest $\left\{\mathcal{T}^f_1, \mathcal{T}^f_2, \ldots, \mathcal{T}^f_n\right\}$, then:

$$F\left(\left\{\mathcal{T}^f_1, \mathcal{T}^f_2, \ldots, \mathcal{T}^f_n\right\}, t\right) = F\left(\mathcal{T}^f_1, t\right) \times \ldots \times F\left(\mathcal{T}^f_n, t\right)$$

The definition is based on Definition 5.8 from [2], but extended from single relations to results of join queries, using Propositions 5.10 and 6.1 from the same paper. It provides a formal specification for the algorithm for constructing f-representations presented in [3] and will allow us to give a formal proof of the correctness of our new algorithm for constructing covers. We remark that the key behind making this specification a practical algorithm is finding a method to efficiently compute the selections $\sigma_{anc(\mathcal{A})=t}R$.

4.2 Covers over F-trees

Since covers are defined with respect to a hypertree decomposition, we need to defined the specific hypertree decomposition that we will use when building covers over f-trees.

Definition 10 (Decomposition associated with an f-tree)**.** For a natural join query Q and f-tree $\mathcal{T}^f$ of Q, we define the (hypertree) decomposition *associated* with $\mathcal{T}^f$ to be the pair $(\mathcal{T}^f, \{B_\mathcal{A}\}_{\mathcal{A}\in\mathcal{T}^f})$, where for each node $\mathcal{A}$ of $\mathcal{T}^f$, $\mathcal{B}_\mathcal{A} = \mathcal{A} \cup anc(\mathcal{A})$, that is, the bag of a node $\mathcal{A}$ is comprised of the node's attribute and its ancestor attributes [2]. Moreover, we define $f(\mathcal{T}^f)$, the *width* of $\mathcal{T}^f$, to be the fractional hypertree witdth of its associated decomposition. Lastly, we define $f(Q)$ to be the minimum width of any f-tree of Q.

It has been shown that $f(Q) = \mathcal{O}(fhtw(Q) \cdot \log|Q|)$ and there exist queries for which this bound is tight [2], i.e. $f(Q) = \Omega(fhtw(Q) \cdot \log|Q|)$. This will play a crucial role in the analysis of *F-Cover*. On the other hand, there exist queries for which $f(Q) = fhtw(Q)$, one instance being the *hierarchical queries* [2], which admit an f-tree with the property that each root-to-leaf path in the f-tree corresponds to the schema of a relation in the query. In Chapter 6 we will see that *F-Cover* performs exceptionally well for such queries. Now let us formally define covers over f-trees:

Definition 11 (Covers over f-trees)**.** Let Q be a natural join query over database $\mathbf{D}$ and $\mathcal{T}^f$ an f-tree of Q. A cover K of the query result $Q(\mathbf{D})$ over $\mathcal{T}^f$ is a cover of $Q(\mathbf{D})$ over the decomposition associated with $\mathcal{T}^f$.

We define $||F||$ to be the total number of singletons in an f-representation F. Then (under data complexity, where the query Q is considered fixed):

Proposition 12 ([2])**.** *For a natural join query Q, f-tree $\mathcal{T}^f$ of Q and database $\mathbf{D}$, it holds that $||\mathcal{T}^f(Q(\mathbf{D}))|| = \mathcal{O}(|\mathbf{D}|^{f(\mathcal{T}^f)})$ and there exist arbitrarily large databases $\mathbf{D}$ for which $||\mathcal{T}^f(Q(\mathbf{D}))|| = \Omega(|\mathbf{D}|^{f(\mathcal{T}^f)})$.*

4.3 An Example

Let us illustrate the definitions with an example. This will also reveal some patterns that will help in describing the computation of the *F-Cover* algorithm. For this example, we will use the same query and relations as in Dataset 1 (presented in Chapter 6), together with a (non-globally consistent) database. We have colored red the singletons in the f-representation $\mathcal{T}^f(Q(\mathbf{D}))$ that correspond to the first tuple in the cover, together with the ranges in $R_1, \dots R_4$ where they can be found. Notice that these ranges are "restricted" at each step, as we traverse $\mathcal{T}^f(Q(\mathbf{D}))$ from top to bottom, until they point to a single tuple in each relation: (a_1, b_1, c_1) in R_1, (a_1, b_1, d_1) in R_2 etc. Also, notice that the two mini-tuples (c_2, d_2) and (c_3, d_3) (colored blue) in the cover K form a cover of the relation represented by the subtree of $\mathcal{T}^f(Q(\mathbf{D}))$ that is colored blue. By "extending" these two tuples with the value b_2, we obtain a cover (over $\mathcal{T}^f_B$) of the relation represented by the subtree of $\mathcal{T}^f(Q(\mathbf{D}))$ that is rooted at the singleton b_2 (colored green).

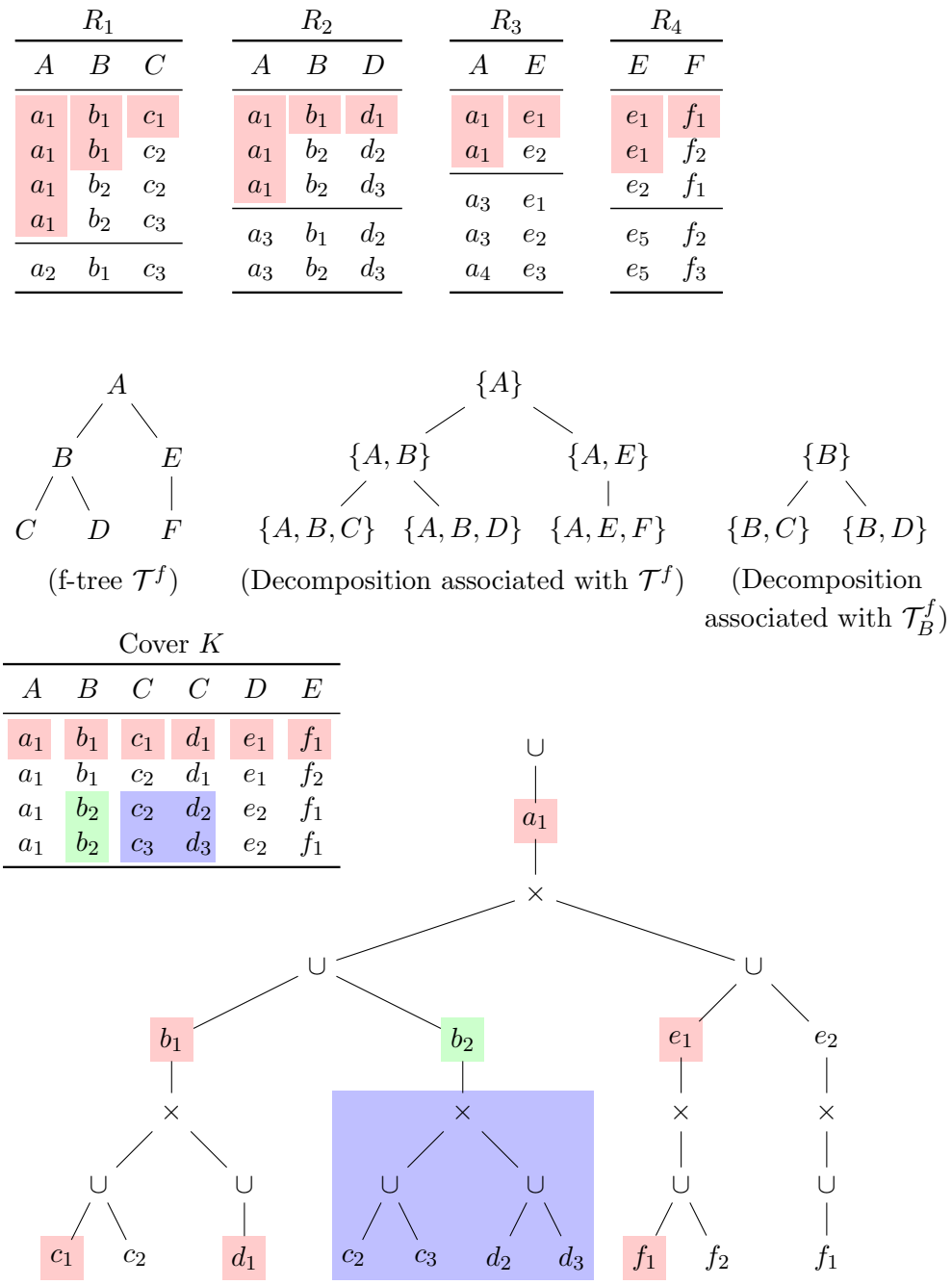

Figure 4.1: Example based on Dataset 1 and the acyclic query Q, the natural join query of relations R_1, R_2, R_3, R_4. Top: Database $\mathbf{D}$, that is not globally consistent. Middle: (left) an f-tree $\mathcal{T}^f$ of Q and its associated decomposition; (right) the decomposition associated with the subtree $\mathcal{T}^f_B$. Bottom: f-representation of $Q(\mathbf{D})$. Please refer to Section 4.3 for detailed explanations of the meaning of the colors.

4.4 The *F-Cover* Algorithm

In this section we present our first monolithic approach to constructing covers of results of join queries. It is based on the idea of "covering" all singletons of an f-representation. The genesis of the algorithm lies in the link between the construction of the f-representation in Definition 9 and the structure of the decomposition associated with an f-representation. The algorithm decomposes the join result one attribute at a time, following the structure of the f-tree. When combining the results from the children of a product node ($\times$) of the f-representation, instead of taking the Cartesian product of the relations corresponding to these children (as one would do when reconstructing the join result from an f-representation of it), we instead take the cover-join of these relations. Intuitively, we are trying to cover all the tuples in these relations with a minimum-size relation, which is exactly what the cover-join of these relations (with disjoint schemas) achieves.

4.4.1 Pseudocode and Proof of Correctness

Figure 4.2 shows the pseudocode of the algorithm. As hinted to previously, the algorithm is based on a particular instantiation of the algorithm from Definition 9 that efficiently computes the selections $\sigma_{anc(\mathcal{A})=context}R$. It achieves this by assuming that the relations are already sorted in a top-down traversal of the f-tree and by keeping track, for each relation R, of two values $(start_R, end_R)$ in $[0..R.size)$ such that

$$R[start_R, end_R] = \sigma_{anc(\mathcal{T}^f)=context}R \tag{4.1}$$

holds at the beginning of each call to *F-Cover*. If $\mathcal{T}^f$ is a forest, then the algorithm computes a cover of the results of calling *F-Cover* recursively on each tree by using a trivial cover-join plan over the linked-list of trees that performs a $foldl1$ of the cover-join operator over the sub-covers. This is essentially the same method we used when constructing covers using cover-join plans in the previous section. The difference is that now these sub-covers are relations over disjoint schemas and hence we can compute a cover-join efficiently by simply matching tuples of the two input sub-covers together, one by one.

If $\mathcal{T}^f$ is not a forest, then we need to compute I. Intersections of sorted columns (relations with one attribute) can be efficiently computed in time linear in the size of the smallest relation [19], which underlies the efficiency of the algorithm (and that of Leapfrog Triejoin) [20]. If $\mathcal{T}^f$ is a leaf, then we simply return I. If $\mathcal{T}^f$ has children, then we call *F-Cover* recursively to compute a cover over the children and extend each tuple of this cover with the value a under consideration. The union of all such extended covers then forms the result returned by the algorithm.

It remains to prove that the result returned by *F-Cover* is indeed a cover as defined in Definition 11. Formally, given a natural join query Q over database $\mathbf{D}$ and f-tree $\mathcal{T}^f$ of Q, we need to prove that *F-Cover* returns a relation K induced by a minimal edge cover of the hypergraph H of $Q(\mathbf{D})$ with respect to the decomposition associated with $\mathcal{T}^f$. In order to do this, we need one more definition:

Definition 13. For a natural join query $Q = R_1 \bowtie \ldots \bowtie R_n$, f-tree $\mathcal{T}^f$ of Q and any subtree or forest $\mathcal{X}$ in $\mathcal{T}^f$, we define the *restriction* of Q to $\mathcal{X}$, denoted $Q_\mathcal{X}$, as $Q_\mathcal{X} = \bowtie_{R \in S(Q): S \cap S(R) \neq \emptyset} \pi_S R$ where S is the set of all attributes that appear in $\mathcal{X}$.

To prove the correctness of *F-Cover*, we will now present an inductive argument that is based on the inductive definition of $\mathcal{T}^f(Q(\mathbf{D}))$ encountered in Definition 9. We will show that the following induction hypothesis holds for all (sub-)f-trees of Q (we omit the *ranges* for brevity):

F-Cover$(context, \_\,, \mathcal{T}^f)$ returns a cover of $\sigma_{anc(\mathcal{T}^f)=context} Q_{\mathcal{T}^f}(\mathbf{D})$ over $\mathcal{T}^f$

For the original f-tree $\mathcal{T}^f$ of Q, this will imply that *F-Cover*$(<>, \_\,, \mathcal{T}^f)$ returns a cover of $Q_{\mathcal{T}^f}(\mathbf{D})$ over $\mathcal{T}^f$. Since $Q_{\mathcal{T}^f} = Q$, the result follows.

- If $\mathcal{T}^f$ is a leaf node $\mathcal{A}$, then, using 4.1, the intersection I computed by *F-Cover* is equal to $\sigma_{anc(\mathcal{A})=context} Q_\mathcal{A}(\mathbf{D})$. Since this relation has only one attribute, I is also the only cover of it and hence the hypothesis holds for leaf nodes.
- If $\mathcal{T}^f$ is a non-empty forest $\left\{\mathcal{T}_1^f, \mathcal{T}_2^f, \ldots, \mathcal{T}_n^f\right\}$, then our algorithm simply follows a cover-join plan to compute a cover over $\mathcal{T}^f$ from the covers over each of the trees $Tf_1, \mathcal{T}_2^f, \ldots, \mathcal{T}_n^f$. Hence, the hypothesis also holds for forests.
- If $\mathcal{T}^f$ has root $\mathcal{A}$ and non-empty set of children $\mathcal{C} = \left\{\mathcal{T}_1^f, \mathcal{T}_2^f, \ldots, \mathcal{T}_n^f\right\}$, then the analysis becomes more interesting. Notice that all bags of the decomposition associated with $\mathcal{T}^f$ contain $\mathcal{A}$ and so all vertices of the hypergraph $H_{\mathcal{T}^f}$ of $\sigma_{anc(\mathcal{T}^f)=context} Q_{\mathcal{T}^f}(\mathbf{D})$ with respect to $\mathcal{T}^f$ will contain some value for $\mathcal{A}$. For $a \in I$, let $H^a_{\mathcal{T}^f}$ be the subgraph of $H_{\mathcal{T}^f}$ obtained by restricting $H_{\mathcal{T}^f}$ to the nodes that contain the value a for $\mathcal{A}$. $H^a_{\mathcal{T}^f}$ will always contain the "singleton" vertex (a). We remove this vertex and denote the resulting subgraph as $H^{a-}_{\mathcal{T}^f}$. Now, by the induction hypothesis, for each value a, *F-Cover*$(context \times a, \_\,, \mathcal{C})$ returns a relation K induced by a minimal edge cover of the hypergraph $H_\mathcal{C}$ of $\sigma_{anc(\mathcal{C})=context \times a} Q_\mathcal{C}(\mathbf{D})$ with respect to $\mathcal{C}$. Observe that, for each value $a \in I$, there exists an isomorphism f_a between $H_\mathcal{C}$ and $H^{a-}_{\mathcal{T}^f}$ that maps each vertex v of $H_\mathcal{C}$ to the vertex $v \cup \{a\}$. Thus, under f_a, the

minimal edge cover of $H_{\mathcal{C}}$ corresponds to a minimal edge cover of $H^{a-}_{\mathcal{T}^f}$. We will mimic f_a and take each tuple of $F\text{-}Cover(context \times a, \_, \mathcal{C}))$ and "extend" it with the value a for $\mathcal{A}$ to obtain a relation that is induced by a minimal edge cover of $H^a_{\mathcal{T}^f}$ (the "singleton" vertex (a) will trivially be included in all of its edges). Now, $H_{\mathcal{T}^f}$ is composed of the set of subgraphs $\left\{H^a_{\mathcal{T}^f} | a \in I\right\}$ and moreover, for distinct values a_1 and a_2 there is no edge between any vertex in $H^{a_1}_{\mathcal{T}^f}$ and vertex in $H^{a_2}_{\mathcal{T}^f}$. Hence we can construct a minimal edge cover of $H_{\mathcal{T}^f}$ as a (set) union of minimal edge covers of $H^a_{\mathcal{T}^f}$, over all values $a \in I$. Therefore we can let: $F\text{-}Cover(context, \_, \mathcal{T}^f) = \bigcup_{a \in I}(a \times F\text{-}Cover(context \times a, \_, \mathcal{C}))$ to obtain a cover of $\sigma_{anc(\mathcal{T}^f)=context} Q_{\mathcal{T}^f}(\mathbf{D})$ over $\mathcal{T}^f$.

To analyse the size of the cover returned by *F-Cover*, we start with a positive observation. Given two relations R_1 and R_2 with disjoint schemas, the cover-join of these relations returns a cover of size equal to the size of the largest relation among R_1 and R_2. Moreover, given a set of relations $S = \{R_1, R_2, \ldots R_n\}$ with disjoint schemas, any cover-join plan of these relations will return a cover of size equal to the largest relation in S, corresponding to a *minimum* edge cover of the hypergraph of the result of their join. By inspecting the proof of the correctness of *F-Cover*, we can see that this result implies that *F-Cover* in fact returns a cover induced by a *minimum* edge cover of the join result. Hence, *F-Cover* performs optimally, but this is with respect to the f-tree given as input. We now turn our attention to finding an upper bound on the size of the cover returned by the algorithm, and to this end, we will break the analysis into two parts.

First, notice that the vertices of the hypergraph of the query result are exactly the set of all distinct paths in $\mathcal{T}^f(Q(\mathbf{D}))$ that start at the root and end at a singleton (not necessarily a leaf). For example, in Figure 4.1, all tuples $(a_1), (a_1, b_1), (a_1, b_2) \ldots (a_1, b_1, c_2) \ldots (a_1, e_2, f_1)$ are vertices of H. For clarity, let us split each such vertex in two: 1) a (possibly empty) *context* tuple and 2) a *singleton* value (the last singleton in the vertex). Following this transformation, the vertices listed above become (context first, singleton second): $((), a_1), ((a_1), b_1), ((a_1), b_2) \ldots ((a_1, b_1), c_2) \ldots ((a_1, e_2), f_1)$. We have thus associated each vertex in H with a singleton value in $\mathcal{T}^f(Q(\mathbf{D}))$.

Secondly, each edge e in a minimal edge cover E of a hypergraph H must contain a vertex v (a *witness*) such that no other edge e' in E contains v. This follows from a simple proof by contradiction. Now consider the cover K returned by *F-Cover*, induced by a minimal edge cover E of the hypergraph H of $Q(\mathbf{D})$ with respect to the decomposition associated with $\mathcal{T}^f$. In the worst case, the edge of H associated with each tuple in K will only have one witness and hence the number of tuples in K can be as big as the number of vertices in H. Since we can associate each such vertex with a singleton in the f-representation $\mathcal{T}^f(Q(\mathbf{D}))$, it follows that the number of tuples in K can be as big as $||\mathcal{T}^f(Q(\mathbf{D}))|| = \Omega(|\mathbf{D}|^{f(\mathcal{T}^f)}) = \Omega(|\mathbf{D}|^{f(Q))}) = \Omega(|\mathbf{D}|^{fhtw(Q)\cdot \log |Q|})$.

F-cover (List<Value> *context*, ranges$[(start_R, end_R)_{R \in S(\mathbf{D})}]$, f-tree $\mathcal{T}^f$)

if $\mathcal{T}^f$ is a non-empty forest $\left\{\mathcal{T}_1^f, \mathcal{T}_2^f, \ldots, \mathcal{T}_n^f\right\}$ **then**:
    $subCovers \leftarrow map$ **F-cover**(context, ranges) $\left\{\mathcal{T}_1^f, \mathcal{T}_2^f, \ldots, \mathcal{T}_n^f\right\}$;
    return $foldl1\ (\mathring{\bowtie})\ subCovers$;

$\mathcal{A}$ = root of $\mathcal{T}^f$; $\quad I = \bigcap_{\substack{R \in S(\mathbf{D}): \\ \mathcal{A} \in S(R)}} \pi_{\mathcal{A}}\left(R[start_R, end_R]\right)$;

if $\mathcal{T}^f$ is a leaf **then**: **return** $\bigcup_{a \in I} a$;

if $\mathcal{T}^f$ has a non-empty set of children $\left\{\mathcal{T}_1^f, \mathcal{T}_2^f, \ldots, \mathcal{T}_n^f\right\}$ **then**:
    $Cover = \emptyset$;
    foreach $a \in I$ **do**:
        foreach $R \in S(\mathbf{D})$ s.t. $A \in S(R)$ **do**:
            restrict ranges $(start_R, end_R)$ to $(start'_R, end'_R) \subseteq (start_R, end_R)$
            s.t. $\pi_A(R[start'_R, end'_R]) = a$;
        $CoverOverChildren \leftarrow$
            F-cover $\left(context \times a, ranges', \left\{\mathcal{T}_1^f, \mathcal{T}_2^f, \ldots, \mathcal{T}_n^f\right\}\right)$;
        if $CoverOverChildren$ is non-empty **then**:
            $ExtendedCoverOverChildren = a \times CoverOverChildren$
            $Cover = Cover \cup ExtendedCoverOverChildren$
    return $Cover$;

Figure 4.2: Pseudocode of the *F-Cover* Algorithm

4.4.2 Implementation Details and Parallelisation

F-Cover assumes that the input relations are sorted in a top-down traversal of the f-tree, which, sequentially, takes $\mathcal{O}(att(Q) \cdot |\mathbf{D}| \cdot \log |\mathbf{D}|)$ time. Given an attribute $\mathcal{A}$, we can clearly sort different relations on $\mathcal{A}$ in parallel, but we can achieve much more. We have exploited the *path condition* that an f-tree must satisfy in order to, additionally, sort sets of relations in parallel. More specifically, for an f-tree $\mathcal{T}^f$ and $\mathcal{A}$ one of its nodes, having children $\mathcal{T}_1^f, \mathcal{T}_2^f, \ldots, \mathcal{T}_n^f$, let S_i be the set of relations having attributes from $\mathcal{T}_i^f$. The path condition implies that these sets are disjoint, and hence we can sort their relations concurrently (sorting each S_i in a preorder traversal of $\mathcal{T}_i^f$). For an f-tree that exhibits high branching, these observations can lower the theoretical bound for sorting the input relations

to $\mathcal{O}(\max(|S(\mathbf{R})| \cdot |\mathbf{R}| \cdot \log |\mathbf{R}|))$, where the maximum is over all relations in the query. In our tests, this step, which is independent of the actual computation of *F-Cover*, always benefited from a parallel implementation. Moreover, the number of cores needed to achieve the theoretical bound is equal to the number of relations in the query, which, in practice, is not very large.

We were hoping to achieve at least similar speed-ups by also computing in parallel the sub-covers (over the children) that the recursive calls to *F-Cover* return. Unfortunately, it turned out that this did not lead to faster computation, at least not consistently. Figure 4.3 shows the time taken by *F-Cover* to compute the results of two join queries (which we will formally define and analyze in Chapter 6) over the Twitter Dataset. Notice that while for Q_5, the parallel computation leads to gains, for Q_3 it leads to a loss in performance. We believe that this is due to the (relatively) small size of each sub-cover and the increased costs associated with spawning multiple threads. Hence, we have decided to sort the input relations in parallel, but to implement *F-Cover* in a sequential manner.

Lastly, we note that when the f-tree used by *F-Cover* has the form of a path (i.e. no branching), the computation of the algorithm is exactly the same as that of the worst-case optimal join algorithm Leapfrog Triejoin [19]. Hence, while *F-Cover* performs optimally in relation to the input f-tree, if this f-tree is a path then the cover it produces (of minimum size) is identical to the full join result.

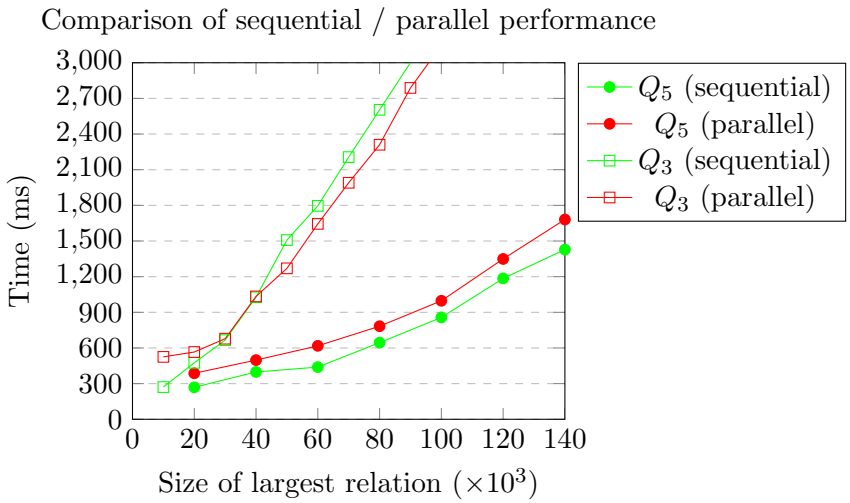

Figure 4.3: Performance of *F-Cover* for two join queries, when the sub-covers over children are computed in parallel versus sequentially.

Chapter 5

Computing Covers over D-trees

We now begin the presentation of the last and most complex algorithm proposed in this paper. Its development originated in the desire to obtain an algorithm that would have the same $\mathcal{O}(Q(|\mathbf{D}|)^{fhtw(Q)})$ bound on the maximum size of a cover it produces as *Plan-Cover*. This is a very important requirement, as maybe the most significant feature of covers is their compactness, which *F-Cover* cannot guarantee. On the other hand, achieving the speed of *F-Cover* was very desirable. By only needing to sort the input relations once, at the beginning (with the efficient type of parallel sorting presented in Section 4.4.2), *D-Cover* does in this sense inherit the best traits of both algorithms presented so far. The presentation in this chapter will be shorter that in the previous chapters, as it builds on a lot of material already encountered. The focus will be explaining the operation of the algorithm and its analysis, including a proof of its correctness.

5.1 D-trees and D-representations

We have already seen an example of a d-tree and a d-representation in Section 1.1. D-representations are very similar to f-representations, the only difference is that we also allow them to reference (point to) already defined d-representations, that is, they use *caching* to avoid representing (and recomputing) existing (sub-)d-representations. This is very similar in style to the concept of common sub-expression elimination from the field of compilers and is what leads to their compactness. Hence, while f-representations are trees, d-representations are directed acyclic graphs in which entire sub-trees can be *shared* to avoid their repetition. Fundamentally, it is this change of structure that calls for a different technique for building covers. Lastly, notice that sharing, done to avoid re-computation, implies the need to actually record that we have already computed a (sub-)d-representation.

For a join query Q, we say that an attribute (node) $\mathcal{A}$ of an f-tree $\mathcal{T}^f$ of Q *depends* on another attribute $\mathcal{B}$ of $\mathcal{T}^f$ if there exists at least one relation in Q whose schema contains both $\mathcal{A}$ and $\mathcal{B}$. Moreover, we say that the subtree $\mathcal{T}^f_{\mathcal{A}}$ depends on a set of attributes S if it holds that, for each attribute C in S, there exists at least one attribute in $\mathcal{T}^f_{\mathcal{A}}$ that depends on C.

Definition 14 (D-trees [3])**.** Given a natural join query Q, a *d-tree* $\mathcal{T}^d$ of Q is a pair $(\mathcal{T}^f, key)$, where $\mathcal{T}^f$ is an f-tree of Q and key is a function mapping each node $\mathcal{A}$ of $\mathcal{T}^f$ to the subset of $anc(\mathcal{A})$ on which $\mathcal{T}^f_{\mathcal{A}}$ depends.

A d-tree records exactly the dependency between attributes and is this information that is leveraged when constructing a d-representation of the join result. More specifically, for a join query Q and an f-tree $\mathcal{T}^f$ of Q, we will say that a node $\mathcal{A}$ is *cacheable* if $key(\mathcal{A}) \neq anc(\mathcal{A})$. Then, for any attribute $\mathcal{B}$ in $anc(\mathcal{A})$ that is not in $key(\mathcal{A})$, the values of $\mathcal{A}$ are independent of the values of $\mathcal{B}$, and hence these $\mathcal{A}$-values can be stored separately and only referenced symbolically for each different $\mathcal{B}$-value. This is the underlying reason why d-representations admit succinct encoding as sets of multimaps [4]. Figure 5.1 (based on Figure 1.1) shows an example of a d-representation of a join result, represented both as a graph and as a set of multimaps. Lastly, we will say that a d-tree *exhibits caching* if it has at least one cacheable node.

Similar to building covers over f-trees, we need to define the particular decomposition that we will use when building covers over d-trees. We give here a set of three definitions that follow the same structure as their corresponding ones in the previous chapter:

Definition 15 (Decomposition associated with a d-tree [2])**.** For a natural join query Q and d-tree $\mathcal{T}^d$ of Q, we define the (hypertree) decomposition *associated* with $\mathcal{T}^d$ to be the pair $(\mathcal{T}^d, \{B_{\mathcal{A}}\}_{\mathcal{A}\in\mathcal{T}^d})$, where for each node $\mathcal{A}$ of $\mathcal{T}^d$, $\mathcal{B}_{\mathcal{A}} = \mathcal{A} \cup key(\mathcal{A})$, that is, the bag of a node $\mathcal{A}$ is comprised of the node's attribute and its key. Moreover, we define $d(\mathcal{T}^d)$, the *width* of $\mathcal{T}^d$, to be the fractional hypertree witdth of its associated decomposition. Lastly, we define $d(Q)$ to be the minimum width of any d-tree of Q.

Definition 16 (Covers over d-trees)**.** Let Q be a natural join query over database $\mathbf{D}$ and $\mathcal{T}^d$ a d-tree of Q. A cover K of the query result $Q(\mathbf{D})$ over $\mathcal{T}^d$ is a cover of $Q(\mathbf{D})$ over the decomposition associated with $\mathcal{T}^d$.

Proposition 17. *For a natural join query Q, d-tree $\mathcal{T}^d$ of Q and database $\mathbf{D}$, the d-representation of $Q(\mathbf{D})$ over $\mathcal{T}^d$ is denoted $\mathcal{T}^d(Q(\mathbf{D}))$. Under data complexity, it holds that the number of singletons in $\mathcal{T}^d(Q(\mathbf{D}))$ is $\mathcal{O}(|\mathbf{D}|^{d(\mathcal{T}^d)})$ and there exist arbitrarily large databases $\mathbf{D}$ for which the number of singletons in $\mathcal{T}^d(Q(\mathbf{D}))$ is $\Omega(|\mathbf{D}|^{d(\mathcal{T}^d)})$ [2].*

We note that the worst-case size of the cover returned by *D-Cover* will be equal to the number of singletons in $\mathcal{T}^d(Q(\mathbf{D}))$ (by a similar argument). It has been shown that, for a query Q, $d(Q) = fhtw(Q)$ [2] and furthermore:

Proposition 18. *For a query Q, d-tree $\mathcal{T}^d$ of Q and database $\mathbf{D}$, $\mathcal{T}^d(Q(\mathbf{D}))$ can be computed (as a set of multi-maps) in time $\tilde{\mathcal{O}}(|\mathbf{D}|^{d(\mathcal{T}^d)})$, by first sorting the input relations in a top-down traversal of the d-tree [4].*

Let us analyze what the multi-maps store. For each attribute (node) $\mathcal{A}$ of $\mathcal{T}^d$, we construct a multi-map $m_{\mathcal{A}}$ that records all the vertices of the hypergraph H of $Q(\mathbf{D})$ that have the schema $\mathcal{A} \cup key(\mathcal{A})$. These are exactly the vertices corresponding to the bag $B_{\mathcal{A}}$ of the decomposition associated with $\mathcal{T}^d$. Hence the set of all entries of all multi-maps is exactly the set of vertices of H. Now, for each entry (key, v) of a multi-map $m_{\mathcal{A}}$, let us *associate* the tuple $key \times v$ with the value v. We have thus associated the vertex $key \times v$ of H with the singleton value v from $\mathcal{T}^d$. Notice that this association is 1 to 1, since the pair $key \times v$ specifies a unique position for the singleton value v in $\mathcal{T}^d$. If we do this across all multi-maps, we have built a 1 to 1 mapping between the vertices of the hypergraph of $Q(\mathbf{D})$ and the singleton values in $\mathcal{T}^d$ (similar to what we did in Section 4.4.1).

Lastly, to explain the computation of *Plan-Cover*, we need one more definition. For a minimal edge cover E and edge e in E, we have defined a *witness* of e to be a vertex $v \in e$ such that no other edge e' in E contains v and showed that each edge in E must in fact have a witness. We will overload this notion in the following way:

Definition 19 (Witness of a tuple)**.** For a natural join query Q, d-tree $\mathcal{T}^d$ of Q, database $\mathbf{D}$, hypergraph H of $Q(\mathbf{D})$ with respect to the decomposition associated with $\mathcal{T}^d$, cover K of $Q(\mathbf{D})$ over $\mathcal{T}^d$ and tuple t of K, we say that a singleton value v of $\mathcal{T}^d$ is a *witness* for t if the vertex of H associated with v is a witness for the edge e of H corresponding to t, with respect to the minimal edge cover of H that induces K.

A simple proof by contradiction reveals that if a relation is a cover, then each one of its tuples has (at least) a witness. Otherwise the cover would not be minimal.

5.2 The *D-Cover* Algorithm

On input a query Q, d-tree $\mathcal{T}^d$ of Q and database $\mathbf{D}$, *D-Cover* first computes $\mathcal{T}^d(Q(\mathbf{D}))$ and stores it as a set of multi-maps. Conceptually, it then performs a combination of breadth-first search and depth-first search over the graph encoding of $\mathcal{T}^d(Q(\mathbf{D}))$ starting from the root (union) node, recording which singleton values it has visited and not considering them again. More specifically, during the computation of *Plan-Cover* a (singleton) value is *fresh*, if it has not yet been visited by the algorithm. Then, fundamentally, *Plan-Cover* visits the fresh values in a breadth-first search approach, and for each visited fresh value, it adds a new tuple to the cover. Moreover, and crucially, this tuple will never be removed from the cover,

that is, it will be part of the final cover returned by the algorithm. The key property that allows *Plan-Cover* to add this new tuple to the final cover is that we can guarantee that this tuple will have at least one witness with respect to the (final) cover returned by the algorithm. This will be formalised in Section 5.2.2.

5.2.1 Pseudocode

We are of the opinion that the computation of *D-Cover* can be better understood by first looking at the computation of the algorithm when the d-tree given as input is a path, i.e. a linked-list. To this end, Figure 5.2 presents a simplified version of the algorithm, that deals with this special case. It is relatively straightforward to generalise the algorithm to the case of arbitrary d-trees, and the general version is given in Figure 5.3. Before we explain the operation of the (general) algorithm, let us quickly note that, while constructing $\mathcal{T}^d(Q(\mathbf{D}))$, we can record, for each singleton value v, the (full) $prefix$ (i.e. assignment for ancestors attributes) that represents the path from v to the root of $\mathcal{T}^d(Q(\mathbf{D}))$.

While we pre-compute and store the d-representation of the join result as a multi-map, the structure behind the computation of *D-Cover* is the (conceptual) graph representation of the d-representation. In this sense, its computation is much easier to understand than the previous algorithms. It visits the fresh singleton values in $\mathcal{T}^d(Q(\mathbf{D}))$ in breadth-first search order, marking each value it visits so that it will not be visited again. In order to add a tuple to the cover for each fresh value c it considers, the algorithm needs to construct the rest of the tuple using singleton values of different attributes. It achieves this using calls to $nextPath$, which is the method that bridges the gap between the multi-map representation and the conceptual traversal of the directed acyclic graph $\mathcal{T}^d(Q(\mathbf{D}))$. We say that a collection of singleton values is *fresh* if it contains at least one fresh value. A call to $\mathcal{A}.nextPath(key)$ returns the next *fresh* root-to-leaf path, in a depth-first seach order, from a subtree $\mathcal{T}'$ of $\mathcal{T}^d(Q(\mathbf{D}))$ rooted at the union ($\cup$ node) of $\mathcal{A}$ values that corresponds to the key $\pi_{key(\mathcal{A})}key$. For example, in Figure 5.1, successive calls to $\mathcal{C}.nextPath(b_1)$ will return the paths (c_1, d_1), (c_2, d_2) and (c_2, d_3) (if c_1, d_1, c_2, d_2, d_3 were all fresh prior to the first call). In fact, in Figure 5.1, this was indeed the case and these were exactly the paths it returned, which can be seen by inspecting the first three tuples of the cover. In addition $nextPath$ also marks the singleton values it visits as visited. Now, when all the singleton values in the subtree $\mathcal{T}'$ have been visited, a call to $\mathcal{A}.nextPath(key)$ cannot return a fresh path anymore, and instead keeps returning the last (rightmost) path. In the previous example, this would be (c_2, d_3). Hence successive calls to $nextPath$ will return fresh paths until there are no more fresh values in the subtree, at which point it will return a non-fresh path.

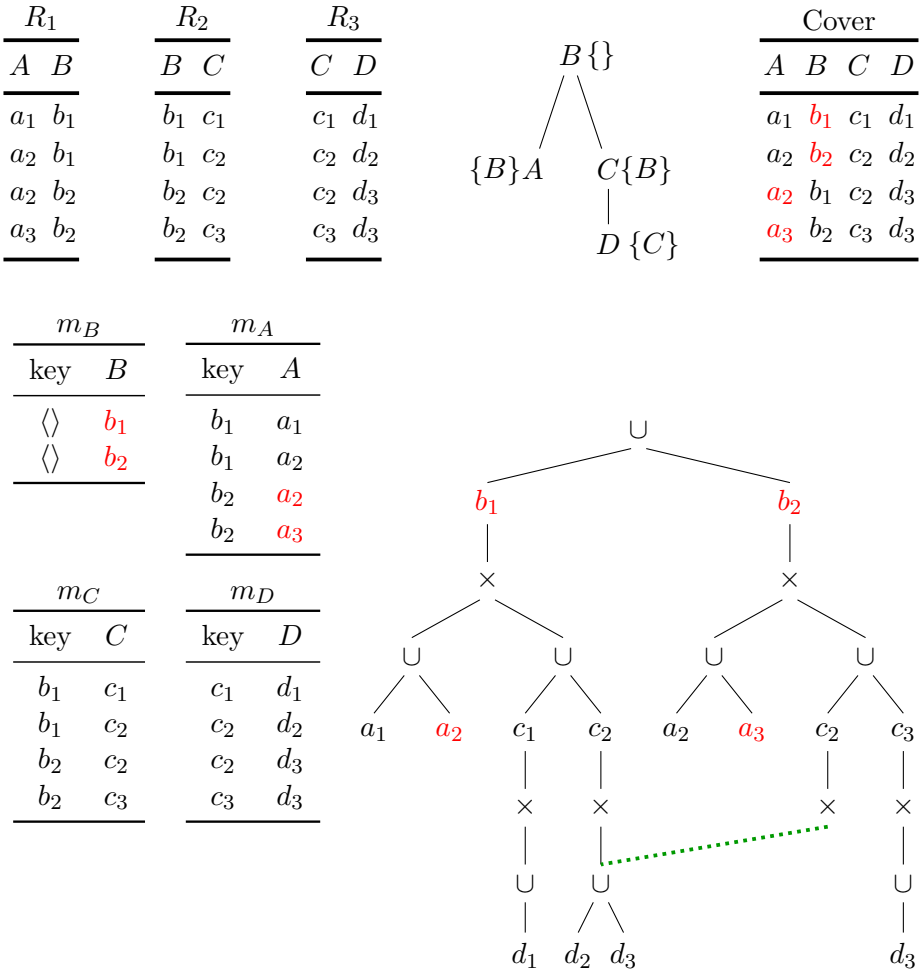

Figure 5.1: Top row: Relations R_1, R_2, R_3, a d-tree of their natural join query and a cover of their natural join result over the given d-tree; Bottom row: (left) the encoding of the d-representation of the join result as a set of multimaps; (right) the same d-representation, encoded as a directed acyclic graph. We have coloured red the (singleton) fresh values v that were marked as visited by *D-Cover*. The rest of the singleton values in the d-representation were marked as visited by calls to *nextPath*. Scanning the cover top-down reveals the order in which these (red) values were considered by *D-Cover*.

D-cover ("path" d-tree $\mathcal{T}^d$)

if $\mathcal{T}^d$ is a non-empty forest $\{\mathcal{T}_1^d, \mathcal{T}_2^d, \ldots, \mathcal{T}_n^d\}$ **then**:
    $subCovers \leftarrow map$ **D-cover** $\{\mathcal{T}_1^d, \mathcal{T}_2^d, \ldots, \mathcal{T}_n^d\}$;
    return $foldl1\ (\overset{\circ}{\bowtie})\ subCovers$;

$Cover = \emptyset$;

foreach node $\mathcal{A}$ **in** $preOrderTraversal(\mathcal{T}^d)$ **do**:
    foreach key k **in** $m_{\mathcal{A}}.keys()$ **do**:
        foreach fresh value v **in** $m_{\mathcal{A}}[k]$ **do**:
            mark v as visited;
            // 1. Get a tuple (any) from $\pi_{anc(\mathcal{A})}\sigma_{\mathcal{A}=v}Q(\mathbf{D})$:
            $prefix = \langle\rangle$;
            if $\mathcal{A}$ is not the root of $\mathcal{T}^d$ **then**:
                $prefix = \mathcal{A}.getPrefix(k)$;
            // 2. Call *nextPath* on the child:
            $nextPathFromChild = \langle\rangle$;
            if $\mathcal{A}$ is not a leaf **then**:
                $child = \mathcal{A}.getChild()$;
                $nextPathFromChild = child.nextPath(prefix \times v)$;
            // 3. Add new tuple to (final) cover:
            $newTuple = prefix \times v \times nextPathFromChild$;
            $Cover = Cover\ \cup newTuple$;

return $Cover$;

Figure 5.2: Pseudocode of the *D-Cover* Algorithm for the special case when $\mathcal{T}^d$ is a path (linked list).

5.2.2 Proof of Correctness

It is clear that *D-Cover* will visit all singleton values in the d-representation, as it performs a breadth-first search over it. Moreover, for each fresh value v (with associated key key), it will put $key \times v$ in the tuple that it adds to the cover. Since there is a one-to-one mapping between singletons v (with associated key key) and vertices $key \times v$ in the hypergraph of the join result, it follows that the relation returned by *D-Cover* will be induced by an edge cover of this hypergraph. It remains to prove that this edge cover is minimal. We will do this by showing that every tuple will have a witness.

As we noted in the previous sections, the reason why *D-Cover* can add a tuple to the final cover and still construct a (minimal) cover is that it knows that its operation will ensure that every tuple it adds to the cover will have at least one witness in the final cover. To see this, note that when the algorithm is considering a particular fresh value v and adds a tuple t (containing v) to the cover, the singleton v is a witness for t. The only issue is that this value may have to be reused, as (part of) the $prefix$ of another singleton value (further down in $\mathcal{T}^d(Q(\mathbf{D}))$) that is added to the cover later on in the computation of *D-Cover*. Crucially, we have defined $nextPath$ in such a way that if it is possible, a call to $nextPath$ will return a fresh path p. Now, by definition, a fresh path contains a fresh value. Let w_2 be the lowest (in $\mathcal{T}^d(Q(\mathbf{D}))$) such value from p. Since this path is added to the tuple together with v, immediately after t is added to the cover, if the path was fresh, then t will in fact have two witnesses (v and w_2) and moreover, since $nextPath$ marked w_2 as visited, w_2 will never be added to any tuple after this point. This is because w_2 is the lowest fresh value in the subtree, which means that the entire subtree below it contained no fresh values, so w_2 is not in danger of being included as part of a $prefix$ of any tuple that is added later by *D-Cover*. Hence if the path was fresh, w_2 will remain the tuple of t for the rest of the operation of the algorithm. Now, if the path returned by $nextPath$ was not fresh, this implies that the entire subtree of $\mathcal{T}^d(Q(\mathbf{D}))$ on which $nextPath$ was called had no fresh values, and hence v, the current (fresh) value under consideration, is not in danger of being contained in any later $prefix$ (this prefix would need to be included with a fresh value that is below v in the subtree rooted at v), and hence in this case v will remain the witness of t until the end. To summarise:

- If v will be reused, as part of a $prefix$ of another tuple added later on to the cover, then w_2 will remain the witness of t.
- If $nextPath$ returned a non-fresh path, then v will remain t's witness.

Hence at least one of v or w_2 will be a witness of t in the final cover returned by the algorithm. Note that it is possible that t will keep both v and w_2 as witnesses until the end. Also, if $nexPath$ returns a path containing many fresh values, they could all remain witnesses for t.

D-cover (d-tree $\mathcal{T}^d$)

if $\mathcal{T}^d$ is a non-empty forest $\{\mathcal{T}_1^d, \mathcal{T}_2^d, \ldots, \mathcal{T}_n^d\}$ **then**:
    $subCovers \leftarrow map$ **D-cover** $\{\mathcal{T}_1^d, \mathcal{T}_2^d, \ldots, \mathcal{T}_n^d\}$;
    return $foldl1\ (\overset{\circ}{\bowtie})\ subCovers$;

$Cover = \emptyset$;

foreach node $\mathcal{A}$ **in** $preOrderTraversal(\mathcal{T}^d)$ **do**:
    foreach key k **in** $m_{\mathcal{A}}.keys()$ **do**:
        foreach fresh value v **in** $m_{\mathcal{A}}[k]$ **do**:
            mark v as visited;
            // 1. Get a tuple (any) from $\pi_{anc(\mathcal{A})}\sigma_{\mathcal{A}=v}Q(\mathbf{D})$:
            $prefix = \langle\rangle$;
            if $\mathcal{A}$ is not the root of $\mathcal{T}^d$ **then**:
                $prefix = \mathcal{A}.getPrefix(k)$;
            // 2. Call *nextPath* on all siblings of ancestors:
            $pathsFromRelatives = \langle\rangle$;
            foreach node $\mathcal{P}$ **in** $anc(\mathcal{T}^d)$ **do**:
                foreach node $\mathcal{R}$ **in** $children(P) \backslash anc(\mathcal{T}^d)$ **do**:
                    $nextPathFromRelative = \mathcal{R}.nextPath(prefix)$;
                    $pathsFromRelatives \mathrel{+}= nextPathFromRelative$;
            // 3. Call *nextPath* on all children:
            $pathsFromChildren = \langle\rangle$;
            if $\mathcal{A}$ is not a leaf **then**:
                foreach node $\mathcal{C}$ **in** $children(\mathcal{A})$ **do**:
                    $nextPathFromChild = child.nextPath(prefix \times v)$;
                    $pathsFromChildren \mathrel{+}= nextPathFromChild$;
            // 4. Add new tuple to (final) cover:
            $newTuple = prefix \times pathsFromRelatives \times v \times pathsFromChildren$;
            $Cover = Cover \cup newTuple$;

return $Cover$;

Figure 5.3: Pseudocode of the *D-Cover* Algorithm that deals with arbitrary d-trees.

Chapter 6

Benchmarks and Analysis

In this chapter we will compare the performance of *Plan-Cover*, *F-Cover* and *D-Cover* in terms of both speed and size of the produced cover. We will use two datasets and a set of 6 different queries, each with different features, covering a wide spectrum of possible join queries. We will comment on the findings and analyse whether or not they matched the results obtained in the theoretical analysis of the algorithms, performed in previous chapters.

6.1 Summary of Findings

Figure 6.1 presents a summary of the best performing algorithms in the benchmarks of Section 6.4, for each of the 6 different queries, together with an aggregation of the results by type of query. *D-Cover* has the overall best performance, though a couple of important observations are in order:

- *Plan-Cover* was not the fastest algorithm in any of the comparisons. After further analysis, we strongly believe that this is due to the fact that, during the computation of the cover join plan, it has to re-sort the intermediary cover result every time it performs a cover-join of it with another bag. On the other hand, the size of the covers it produces is always at most twice as large as that coming from the best algorithm.
- *F-Cover* has very high throughput, helped in part by the lightweight nature of its computation. After initially sorting the input relations, it does not need to do any other sorting during its entire operation. Also, the size of the covers it returns is very sensitive to the width of the f-tree given as input, which is in line with our theoretical analysis.
- *D-Cover* always produced the smallest cover, even though it shares the same (small) theoretical upper bound with *Plan-Cover*. Additionally, it was always either the fastest algorithm or just a constant factor away from it. It thus combines these two desired features and is in this sense the most reliable among all algorithms analysed.

6.2 Queries and Datasets

We will give a description of each query and explain why it was chosen. Q_1 uses Dataset 1 (described in detail in Figure 4.1), which was provided by the FDB Group at Oxford, while the other queries use the Twitter Dataset [22], which is very flexible, containing one binary relation $Friends$. This allowed us to build a wide array of queries by composing it. For each query, we will define a d-tree and construct an f-tree by discarding the keys of the former. Lastly, *Plan-Cover* will be given as input the decomposition associated with the d-tree.

Q_1 This is the acyclic query presented in Figure 4.1, running on Dataset 1: $Q_1 = R_1(A.B, C) \bowtie R_2(A, B, D) \bowtie R_3(A, E) \bowtie R_4(E, F)$. It was chosen because it is close to being hierarchical, but in the same time its d-tree exhibits some caching (attribute F depends only on E).

Q_2 This is an acyclic query whose d-tree does not exhibit caching. In particular, it is a hierarchical query: $Q_2 = R_1(A, B) \bowtie R_2(A, C) \bowtie R_3(A, D)$. It was very important to assess the difference in performance between *D-Cover* and *F-Cover* for queries for which $f(Q) = fhtw(Q) = 1$, where the lightweight nature of *F-Cover* make it a clear winner. As the plots suggest, there is indeed a (small) hidden *constant* factor between the two algorithms.

Q_3 This is an acyclic, (non-hierarchical) extension of Q_2 that adds caching: $Q_3 = R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(A, D) \bowtie R_4(D, E) \bowtie R_5(A, F) \bowtie R_6(F, G)$ As the plots reveal, when caching is involved, the size of the cover produced by *F-Cover* increases tremendously, while *D-Cover* performs very well.

Q_4 This type of query is part of a category known in the literature as "path queries" [4]: $Q_4 = R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(C, D) \bowtie R_4(D, E)$. The particular d-tree used for this query also has the structure of a path, which differentiates this query from Q_2 and Q_3, which have a very balanced d-tree. Here *F-Cover* becomes equivalent to an implementation of the Leapfrog Triejoin join algorithm and hence it computes the actual join result.

Q_5 This is the (cyclic) bow tie query, first encountered in Figure 3.1: $Q_5 = R_1(A, B) \bowtie R_2(B, C) \bowtie R_3(A, C) \bowtie R_4(A, D) \bowtie R_5(D, E) \bowtie R_6(A, E)$. It reveals the performance impact that materialising the bags has on *Plan-Cover*. The results suggest that the other two algorithms are very well suited for cyclic queries.

Q_6 This is a cyclic query whose d-tree exhibits caching. This represents an important combination of features, and the results show that, in fact, the hierarchy between algorithms changes compared to Q_5.

	Hierarchical	Caching	Cyclic	Smallest Cover	Fastest
Q_1	almost	little	no	D	F
Q_2	yes	no	no	All equal	F
Q_3	no	yes	no	D	D
Q_4	no	yes	no	F & D	D
Q_5	no	no	yes	All equal	F
Q_6	no	yes	yes	D	D
Winner	✓			D	F
/Category		✓		D	D
			✓	D	F & D
Overall Winner				***D***	***F*** & ***D***

Figure 6.1: A summary of the best performing algorithms among *F-Cover* (F), *D-Cover* (D) and *Plan-Cover* for the queries Q_1 to Q_6 and a breakdown by category (type of query). We use "All equal" to mean that all three algorithms produced covers of equal size and we use the notation F & D to mean that *F-Cover* and *D-Cover* tied for first place.

6.3 Setup

To construct the plots, we ran each algorithm 5 times for each data point and averaged the results. The databases were loaded in memory prior to performing the benchmarks.

- For Dataset 1, there are 10 data points, corresponding to 10 databases, of increasing size, with four relations over the same schema.
- For the Twitter Dataset [22], we had access to a large relation *Friends* with two attributes. We created multiple database instances by varying the number of tuples read from *Friends* (into a relation instance) and mapping all relation symbols in the join queries to this same relation instance. Hence, all relations in the queries have schemas of exactly two attributes and refer to relations of equal size.

The system we performed the benchmarking on is a MacBook Pro with the following specifications:

- Processor: 2.3GHz dual-core Intel Core i5, Turbo Boost up to 3.6GHz, with 64MB of eDRAM
- Memory: 8 GB 2133 MHz LPDDR3
- Graphics: Intel Iris Plus Graphics 640 1536 MB
- Operating System: macOs 10.13.4 High Sierra

6.4 Benchmarks

For each one of the queries Q_1 to Q_6, there are two plots: one showing the time taken by each of the three algorithms, and the other one showing the size of the resulting covers (number of tuples). All plots are drawn by varying the number of tuples in the largest relation in the database.

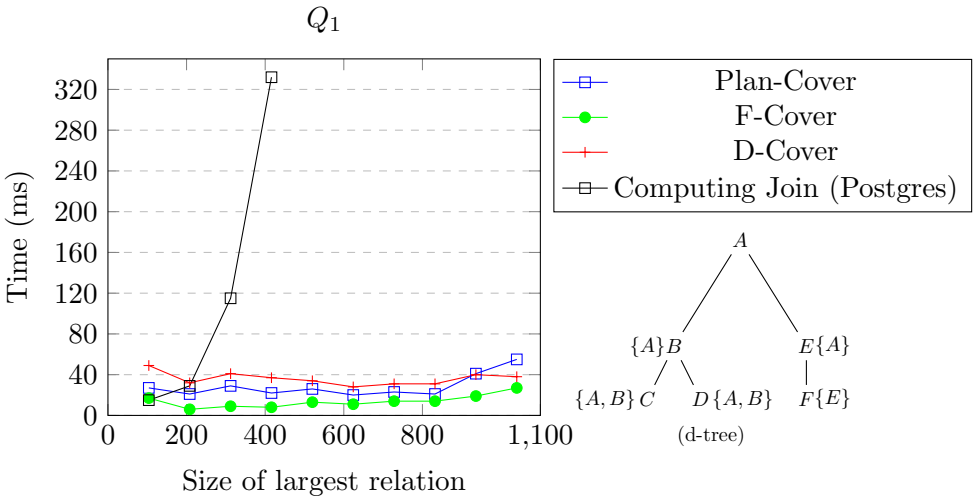

Figure 6.2: Acyclic, almost hierarchical query, with little caching.

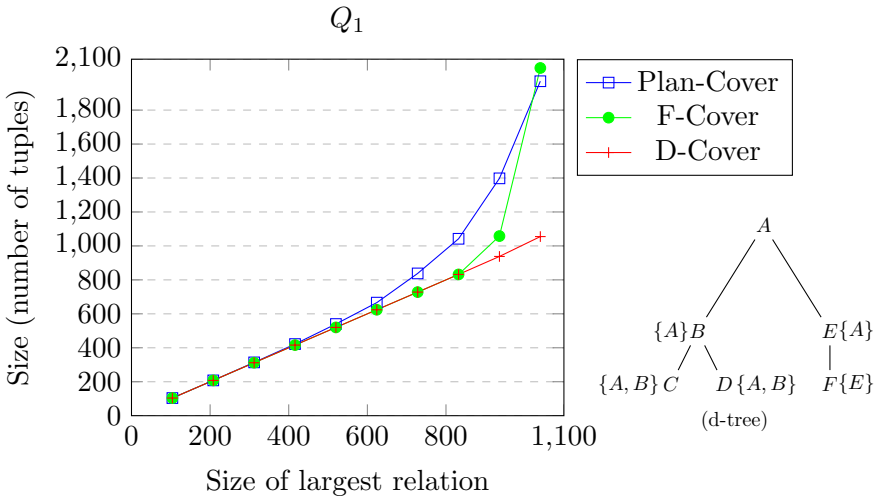

Figure 6.3: Acyclic, almost hierarchical query, with little caching.

We initially tried to include the computation of the join result by Postgres in all plots, but as it turned out, the data points obtained from it were way outside the plots, as it took much longer to compute the full join result than the covers, and, as expected, the former was exponentially larger than the covers.

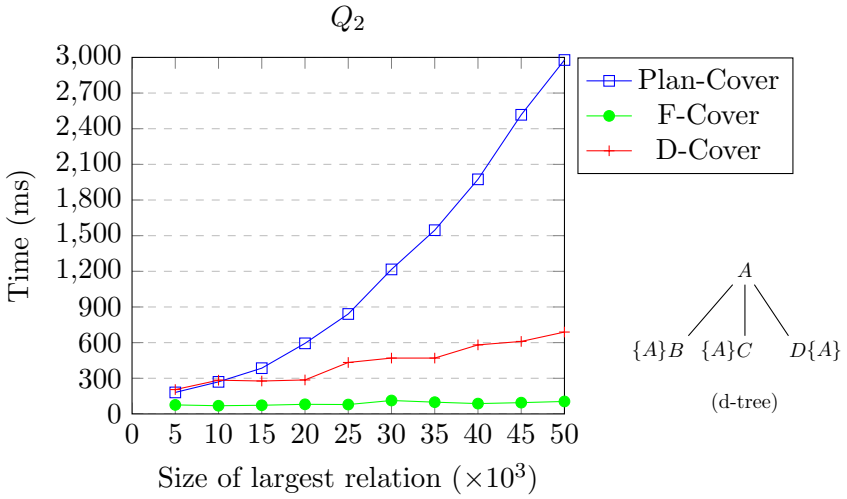

Figure 6.4: Acyclic, hierarchical query, with no caching.

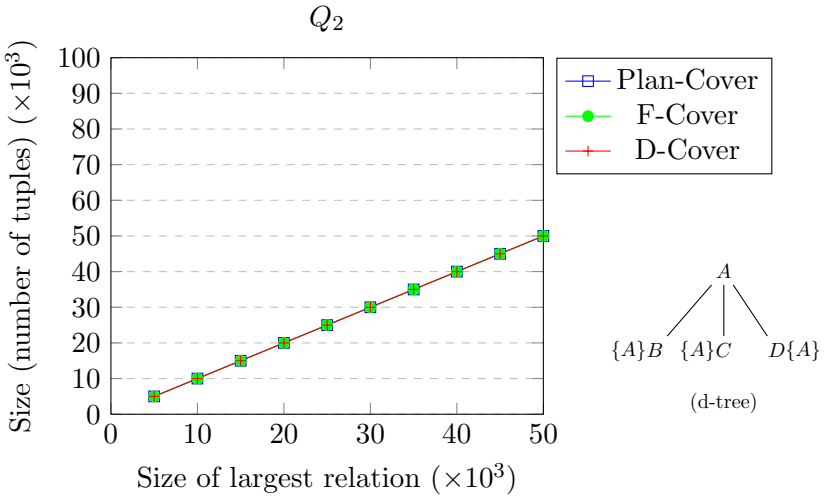

Figure 6.5: Acyclic, hierarchical query, with no caching. All three algorithms produce covers of equal size.

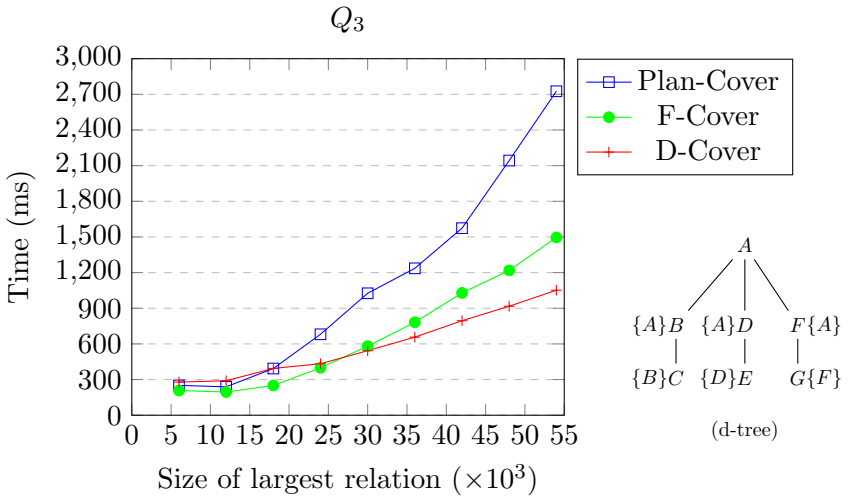

Figure 6.6: Acyclic, non-hierarchical query, whose d-tree exhibits caching.

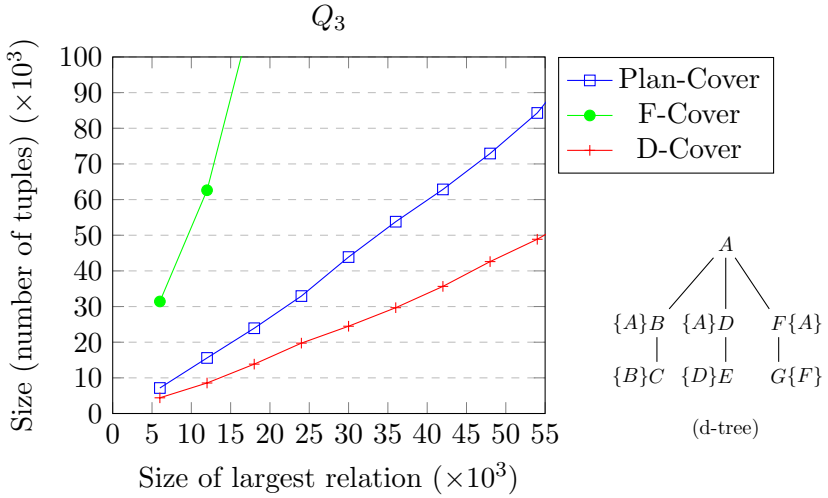

Figure 6.7: Acyclic, non-hierarchical query, whose d-tree exhibits caching.

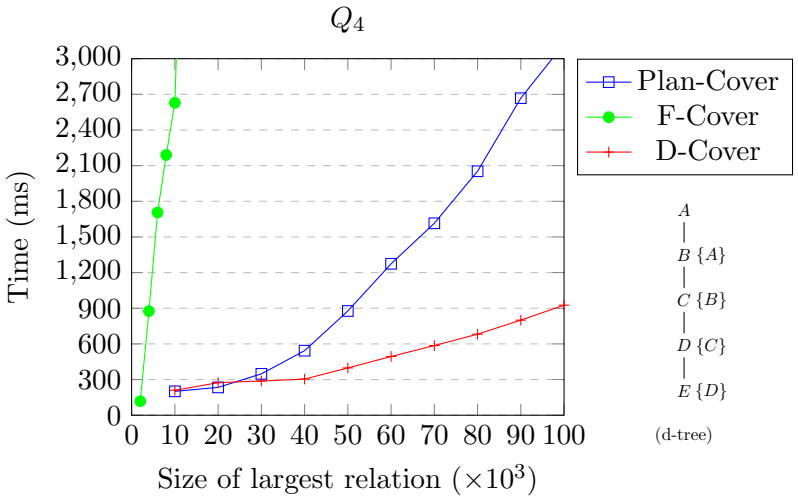

Figure 6.8: The "path query": acyclic, non-hierarchical, its d-tree exhibits caching.

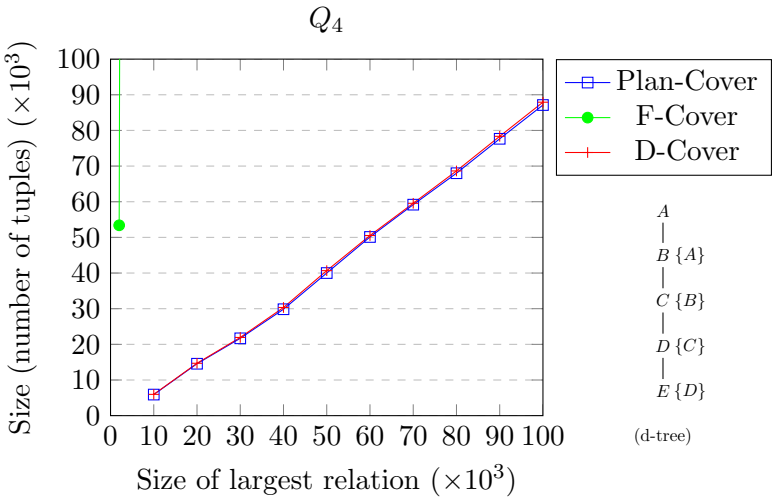

Figure 6.9: The "path query": acyclic, non-hierarchical, its d-tree exhibits caching. *Plan-Cover* and *D-Cover* produce covers of equal size.

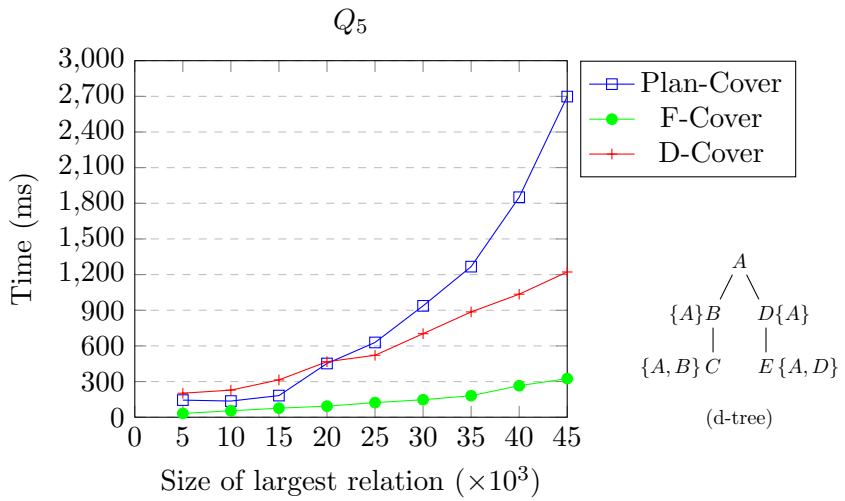

Figure 6.10: The "bow tie" query: non-hierarchical, cyclic, no caching.

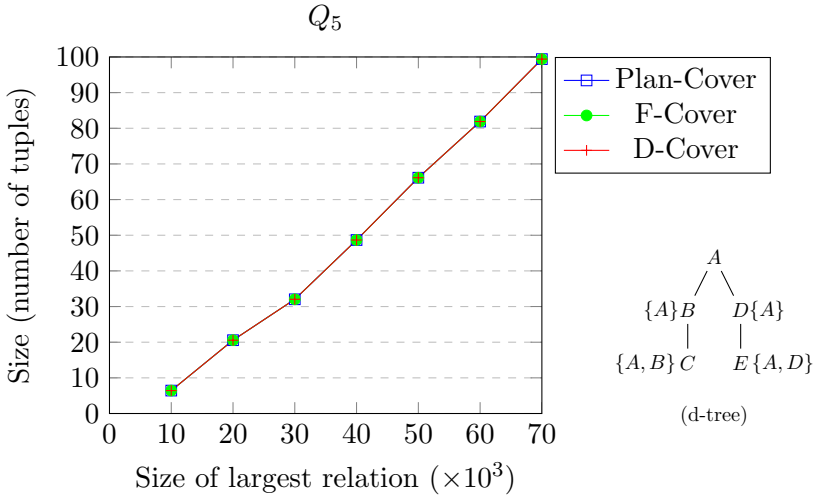

Figure 6.11: The "bow tie" query: non-hierarchical, cyclic, no caching.

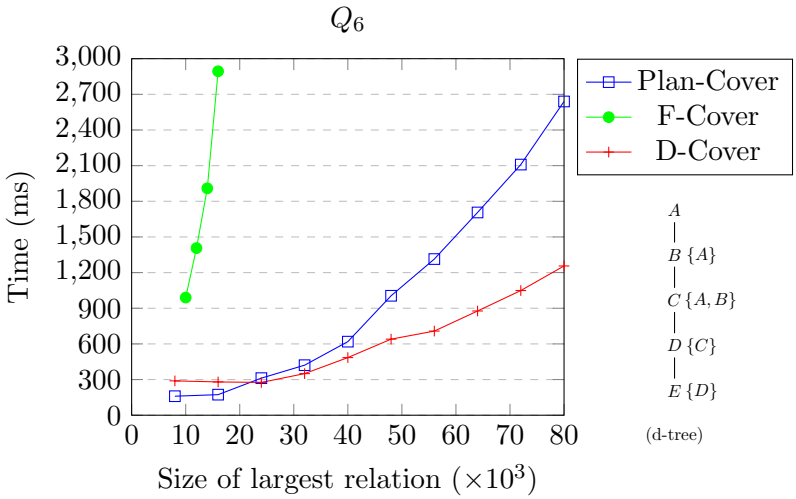

Figure 6.12: Cyclic, non-hierarchical query, whose d-tree exhibits caching.

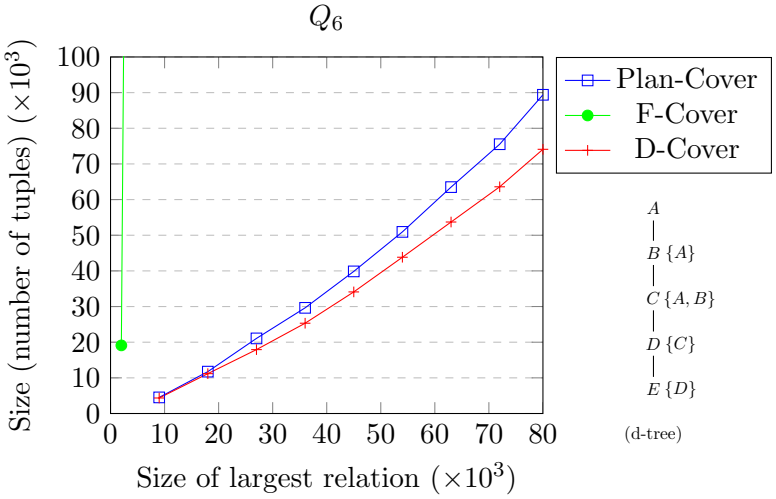

Figure 6.13: Cyclic, non-hierarchical query, whose d-tree exhibits caching.

Chapter 7

Personal Thoughts and Directions for Future Work

This project stemmed from the desire to assess the performance of *Plan−Cover* and try to find a method that will leverage factorised representations to build covers. After finishing and testing *F-Cover*, we were incredibly excited to see that it is exceptionally fast, especially compared to *Plan-Cover*. But the size of the some of the covers returned by the former made them unusable with regard to the fundamental trait that a cover should possess: succinctness. Hence, we set out to combine the traits of the two algorithms (small theoretical upper bound on size of covers returned by *Plan-Cover* and speed of *F-Cover*). Here is where the fun part really began. The development of *D-Cover* was stopped multiple times, as we were unable to arrive at a solution and began questioning whether one (of the form we were looking for) actually exists. I have to admit that finding a construction for *D-Cover* and witnessing its performance was one of my most rewarding academic achievements yet, and I do not plan to throw away the tens of pages of drawings of d-representations anytime soon. It thoroughly enjoyed the research that went into this project and I wish I had more time, as there are a couple of very interesting paths the project could continue on. In particular, whether or not a parallel implementation of the computation of *F-Cover* leads to performance gains remained unanswered, and we believe that a closer analysis would be highly beneficial. Moreover, while in our implementation *D-Cover* first computes a d-representation and then operates on it, this is not strictly necessary for the algorithm and the values at each union node in the d-representation could be computed on the fly, one at a time, as they are needed. While this would not lead to asymptotically smaller covers, it is definitely still worth implementing. Furthermore, parallel computation could again be leveraged to potentially improve the performance of *D-Cover*.

Chapter 8

Code Listings

As this project has more than 5000 lines of code, a full listing of all the code would not fit the allowed number of extra pages, which is currently set by the Department at 40. We are the opinion that including a selection of files is still worth it, so we have decided to pick the ones containing the core functionality of the system. In particular, we will list:

- *StatsEngine*: This class contains the code to run the benchmarks and obtain the datapoints that were then plotted.
- *HTEngine*: This class provides the core structure of the computation of *Plan-Cover*.
- *FTreeEngine*: This class contains the implementation and almost all the functionality needed by *F-Cover*.
- *DTreeEngine*: This class comprises the core logic of *D-Cover* and also includes methods to build a d-representation from the input relations.

8.1 StatsEngine

```
package statistics;

import algorithms.DTreeEngine;
import algorithms.FTreeEngine;
import datamodel.*;
import algorithms.HTEngine;
import datasets.*;

import java.util.*;
import java.util.stream.Collectors;

public class StatsEngine
{
    public static final int numberOfIterations = 5;

    public static final int maxAllowedTimePerExperiment = 3200; // (ms)
    public static final int maxAllowedSizePerExperiment = 5000000; // (#tuples)
```

```
public static Map<String, Boolean> areOn =
                        new HashMap<>(Map.of("HTree", true,
                                             "FTree", true,
                                             "DTree", true));


public static void computePlot()
{
    Dataset dataset = new TwitterBowtie();

    System.out.println("\nFor dataset: " + dataset.getName());

    int howManyAreOn = (int) areOn.values().stream()
            .filter(Boolean::booleanValue)
            .count();

    // Load databases in memory:
    List<List<Database>> databases =
            dataset.getDatabases(howManyAreOn * numberOfIterations,
                                 10000,
                                 100000,
                                 10000);

    TreeNode hTree = dataset.getHTree();
    List<FTreeNode> fTree = dataset.getFTree();

    var resultsTimeHTree = new HashMap<Integer, Long>();
    var resultsTimeDTree = new HashMap<Integer, Long>();
    var resultsTimeFTree = new HashMap<Integer, Long>();

    var resultsSizeHTree = new HashMap<Integer, Integer>();
    var resultsSizeDTree = new HashMap<Integer, Integer>();
    var resultsSizeFTree = new HashMap<Integer, Integer>();

    for (List<Database> instancesOfEqualSize : databases)
    {
        // |instancesOfEqualSize| is a list of identical databases,
        // so for each algorithm, the experiments are repeated on
        // fresh, unsorted, identical, databases.

        System.out.println(
                "\nDatabase " + instancesOfEqualSize.get(0).getName() + ":");

        var maxRelSize = instancesOfEqualSize.get(0).maximumRelationSize();

        if (areOn.get("HTree"))
        {
            var experimentsHTree = new LinkedList<Experiment>();

            for (int i = 0; i < numberOfIterations; i++)
            {
                Database database = instancesOfEqualSize.remove(0);
```

```
            Experiment experiment =
                    () -> HTEngine.computeCover(database,
                                                hTree,
                                                false);
            experimentsHTree.add(experiment);
        }

        Statistics statsHTree = averageExperiments(experimentsHTree,
                                                   numberOfIterations);

        resultsTimeHTree.put(maxRelSize, statsHTree.getTime());
        resultsSizeHTree.put(maxRelSize, statsHTree.getSize());

        System.out.println("\nOver HTree: " + statsHTree);

        if (statsHTree.getTime() > maxAllowedTimePerExperiment
                || statsHTree.getSize() > maxAllowedSizePerExperiment)
        {
            areOn.replace("HTree", false);
        }

    }

    if (areOn.get("FTree"))
    {
        var experimentsFTree = new LinkedList<Experiment>();

        for (int i = 0; i < numberOfIterations; i++)
        {
            Database database = instancesOfEqualSize.remove(0);

            Experiment experiment =
                    () -> FTreeEngine.computeCover(database,
                                                   fTree,
                                                   false,
                                                   false,
                                                   true);
            experimentsFTree.add(experiment);
        }


        Statistics statsFTree = averageExperiments(experimentsFTree,
                                                   numberOfIterations);
        resultsTimeFTree.put(maxRelSize, statsFTree.getTime());
        resultsSizeFTree.put(maxRelSize, statsFTree.getSize());

        System.out.println("\nOver FTree: " + statsFTree);

        if (statsFTree.getTime() > maxAllowedTimePerExperiment
                || statsFTree.getSize() > maxAllowedSizePerExperiment)
        {
            areOn.replace("FTree", false);
        }
    }
```

```
    if (areOn.get("DTree"))
    {
        var experimentsDTree = new LinkedList<Experiment>();

        for (int i = 0; i < numberOfIterations; i++)
        {
            Database database = instancesOfEqualSize.remove(0);

            Experiment experiment =
                () -> DTreeEngine.computeCover(database,
                                               () -> dataset.getNewDTree(),
                                               false,
                                               true);

            experimentsDTree.add(experiment);
        }


        Statistics statsDTree = averageExperiments(experimentsDTree,
                                                   numberOfIterations);
        System.out.println("\nOver DTree: " + statsDTree);

        resultsTimeDTree.put(maxRelSize, statsDTree.getTime());
        resultsSizeDTree.put(maxRelSize, statsDTree.getSize());

        if (statsDTree.getTime() > maxAllowedTimePerExperiment
                || statsDTree.getSize() > maxAllowedSizePerExperiment)
        {
            areOn.replace("DTree", false);
        }
    }
}


System.out.println("\nTime: ");
System.out.println(
        "For Plan-Cover: " + resultsTimeHTree.entrySet().stream()
                          .sorted(Map.Entry.comparingByKey())
                          .map(entry -> "(" + entry.getKey() / 1000
                                  + "," + entry.getValue() + ")")
                          .collect(Collectors.joining("")));

System.out.println(
        "For D-Cover: " + resultsTimeDTree.entrySet().stream()
                        .sorted(Map.Entry.comparingByKey())
                        .map(entry -> "(" + entry.getKey() / 1000
                                + "," + entry.getValue() + ")")
                        .collect(Collectors.joining("")));

System.out.println(
        "For F-Cover: " + resultsTimeFTree.entrySet().stream()
                          .sorted(Map.Entry.comparingByKey())
                          .map(entry -> "(" + entry.getKey() / 1000
```

```
                                        + "," + entry.getValue() + ")")
                                .collect(Collectors.joining("")));

        System.out.println("\nSize: ");
        System.out.println(
                "For Plan-Cover: " + resultsSizeHTree.entrySet().stream()
                        .sorted(Map.Entry.comparingByKey())
                        .map(entry -> "(" + entry.getKey() / 1000
                                + "," + (float) entry.getValue() / 1000 + ")")
                        .collect(Collectors.joining("")));

        System.out.println(
                "For D-Cover: " + resultsSizeDTree.entrySet().stream()
                        .sorted(Map.Entry.comparingByKey())
                        .map(entry -> "(" + entry.getKey() / 1000
                                + "," + (float) entry.getValue() / 1000 + ")")
                        .collect(Collectors.joining("")));

        System.out.println("For F-Cover: " + resultsSizeFTree.entrySet().stream()
                        .sorted(Map.Entry.comparingByKey())
                        .map(entry -> "(" + entry.getKey() / 1000
                                + "," + (float) entry.getValue() / 1000 + ")")
                        .collect(Collectors.joining("")));
    }

    public static Statistics averageExperiments(List<Experiment> experiments,
                                                int numberOfIterations)
    {
        long startTime = System.nanoTime();

        DynamicRelation result = null;

        for (Experiment experiment : experiments)
        {
            result = experiment.compute();
        }

        long endTime = System.nanoTime();

        long averageTimeMs =
                (endTime - startTime) / (numberOfIterations * 1000000);
        int numberOfTuples = result.size();

        return new Statistics(averageTimeMs, numberOfTuples);
    }


    public static void main(String[] args)
    {
        computePlot();
    }
}
```

8.2 HTEngine

```
package algorithms;

import datamodel.*;

import java.util.stream.Collectors;

public class HTEngine
{
    public static DynamicRelation computeCover(Database database,
                                               TreeNode root,
                                               boolean shouldPrint)
    {
        // === 1 === Compute the bags

        root.preOrderTraversal()
                .forEach(treeNode -> treeNode.materialiseBag(database));


        // === 2 === Make the bags globally consistent
        // Use two passes through the tree, first bottom-up and then top-down:

        root.postOrderTraversal()
                .filter(treeNode -> !treeNode.isRoot())
                .forEachOrdered(TreeNode::parentLeftSemiJoinThis);
        root.preOrderTraversal()
                .filter(treeNode -> !treeNode.isRoot())
                .forEachOrdered(TreeNode::thisLeftSemiJoinParent);


        // === 3 === Compute a cover of the bags using the cover-join plans
        // We want to implement the following:
        // fold1 (this::binaryCoverJoin) preOrderTraversal

        RelationInstance cover = root.preOrderTraversal()
                .sequential()
                .map(TreeNode::getBag)
                .reduce(JoinAlgorithms::binaryCoverJoin)
                .get();

        if (shouldPrint)
        {
            System.out.println(
                    "\nThe cover constructed using cover-join plans is:\n"
                            + cover);
        }

        return new DynamicRelation(cover);
    }
}
```

8.3 FTreeEngine

```
package algorithms;

import datamodel.*;
import util.Range;

import java.util.*;
import java.util.concurrent.*;
import java.util.function.Function;
import java.util.stream.Collectors;
import java.util.stream.Stream;


public class FTreeEngine
{
    public static Cover<FTreeNode> computeCover(Database database,
                                                List<FTreeNode> fTree,
                                                boolean shouldPrint,
                                                boolean parallel,
                                                boolean parallelSort)
    {
        if (parallelSort)
        {
            database.parallelSortAccordingToTree(fTree);
        }
        else
        {
            // Sort database according to getAttributes in a
            // pre-order traversal of ftree:
            fTree.stream()
                .flatMap(FTreeNode::preOrderTraversal)
                .map(FTreeNode::getAttribute)
                .collect(Collectors.toCollection(LinkedList::new))
                .descendingIterator()
                .forEachRemaining(attribute ->
                        database.relationsWith(attribute)
                                .forEach(rel -> rel.sortBy(attribute)));
        }

        // Initialise ranges:
        Map<RelationInstance, Range> ranges =
                database.relations()
                        .collect(Collectors.toMap(Function.identity(),
                                                  Range::new));

        Cover cover = recComputeCover(fTree, ranges, parallel);

        if (shouldPrint)
            System.out.println("\nThe cover is:\n" + cover);

        return cover;
    }
```

```
public static Cover<FTreeNode> recComputeCover(
        List<FTreeNode> fTree,
        Map<RelationInstance, Range> ranges,
        boolean parallel)
{
    // If f-tree is a forest:
    if (fTree.size() > 1)
    {
        if (parallel)
        {
            ExecutorService executorService =
                    Executors.newFixedThreadPool(fTree.size());

            Stream<Callable<Cover<FTreeNode>>> subCovers = fTree.stream()
                    .map(child -> () -> recComputeCover(List.of(child),
                                                        ranges,
                                                        parallel));
            try
            {
                List<Future<Cover<FTreeNode>>> results =
                        executorService.invokeAll(
                                subCovers.collect(Collectors.toList()));
                Stream<Cover<FTreeNode>> coversOfChildren =
                        results.stream().map(
                                res -> {try {return res.get();}
                                catch (Exception e) {}
                                    return null;});
                executorService.shutdownNow();

                return JoinAlgorithms.coverJoinOverDisjointSchemas(
                            coversOfChildren);
            }
            catch (InterruptedException e)
            {
                System.out.println(e.getStackTrace());
            }
        }
        else
        {
            Stream<Cover<FTreeNode>> coversOfChildren = fTree.stream()
                    .map(child -> recComputeCover(List.of(child),
                                                  ranges,
                                                  parallel));

            return JoinAlgorithms.coverJoinOverDisjointSchemas(
                        coversOfChildren);
        }
    }

    var result = new Cover<>("Cover", fTree);

    FTreeNode root = fTree.get(0);

    Attribute currentAttribute = root.getAttribute();
```

```
        Map<Integer, Map<RelationInstance, Range>> valuesAndTheirRanges =
            JoinAlgorithms.intersection(currentAttribute, ranges);

        List<FTreeNode> children = root.getChildren();

        if (children.size() == 0)
        {
            // |current| is a leaf node
            valuesAndTheirRanges.keySet().stream()
                    .sorted()
                    .forEachOrdered(
                        value -> result.addSingleton(currentAttribute, value));
        }
        else
        {
            // |current| is an inner node
            for (Map.Entry<Integer, Map<RelationInstance, Range>> valueAndRanges :
                    valuesAndTheirRanges.entrySet().stream()
                            .sorted(Map.Entry.comparingByKey())
                            .collect(Collectors.toList()))
            {
                int value = valueAndRanges.getKey();

                var restrictedRanges = valueAndRanges.getValue();
                ranges.forEach(restrictedRanges::putIfAbsent);

                Cover coverOverChildren = recComputeCover(children,
                                                          restrictedRanges,
                                                          parallel);

                if (coverOverChildren.size() > 0)
                {
                    coverOverChildren.extendWithValueForAttribute(currentAttribute,
                                                                  value);
                    result.addAllTuplesFrom(coverOverChildren);
                }
            }
        }

        return result;
    }
}
```

8.4 DTreeEngine

```
package algorithms;

import datamodel.*;
import util.Range;

import java.util.*;
import java.util.function.Function;
import java.util.function.Supplier;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class DTreeEngine
{
    public static Cover<DTreeNode> computeCover(
            Database database,
            Supplier<List<DTreeNode>> dTreeSupplier,
            boolean shouldPrint,
            boolean parallelSort)
    {
        List<DTreeNode> dTree = dTreeSupplier.get();

        if (parallelSort)
        {
            database.parallelSortAccordingToTree(dTree);
        }
        else
        {
            // ==1== Sort relations in pre-order traversal of d-tree:
            dTree.stream()
                    .flatMap(DTreeNode::preOrderTraversal)
                    .map(DTreeNode::getAttribute)
                    .collect(Collectors.toCollection(LinkedList::new))
                    .descendingIterator()
                    .forEachRemaining(attribute ->
                          database.relationsWith(attribute)
                                  .forEach(rel -> rel.sortBy(attribute)));
        }


        // Initialise ranges:
        Map<RelationInstance, Range> ranges =
                database.relations()
                        .collect(Collectors.toMap(Function.identity(),
                                                  Range::new));


        // ==2== Compute d-representation from database input relations
        recComputeDrepFromDatabase(dTree, new HashMap<>(), ranges);
```

```
    // ==3== Make all values fresh
            dTree.forEach(tree ->
                    tree.preOrderTraversal()
                        .forEach(DTreeNode::makeAllValuesFresh));

    // ==4== Compute cover from d-representation
    Cover<DTreeNode> result = computeCoverFromDrep(dTree, shouldPrint);

    if (shouldPrint)
        System.out.println("\n" + result);

    return result;
}

/**
 * Computes a d-representation, stored as multimaps on D-tree nodes
 * @param dTree the dtree (possibly a forest)
 * @param varMap mapping for the ancestor attributes
 * @param ranges ranges (in relations) to look in
 */
public static boolean recComputeDrepFromDatabase(
        List<DTreeNode> dTree,
        Map<Attribute, Integer> varMap,
        Map<RelationInstance, Range> ranges)
{
    if (dTree.size() > 1)
    {
        Map<Boolean, List<DTreeNode>> nodesPartition = dTree.stream()
                .collect(Collectors.partitioningBy(
                        node -> recComputeDrepFromDatabase(List.of(node),
                                                           varMap,
                                                           ranges)));

        List<DTreeNode> falseNodes = nodesPartition.get(false);

        return falseNodes.isEmpty();
    }

    DTreeNode root = dTree.get(0);

    Attribute currentAttribute = root.getAttribute();

    if (root.hasKeyAssignment(varMap))
    {
        // This node is cacheable and entire subtree of the d-rep
        // rooted at it has already been computed
        return true;
    }

    Map<Integer, Map<RelationInstance, Range>> valuesAndTheirRanges =
            JoinAlgorithms.intersection(currentAttribute, ranges);

    List<DTreeNode> children = root.getChildren();
```

```
    if (children.size() == 0)
    {
        // this node is a leaf node

        if (valuesAndTheirRanges.isEmpty())
            return false;

        // Construct multimap for the attribute of this D-tree node:
        valuesAndTheirRanges.keySet().stream()
                .sorted()
                .forEachOrdered(valueForAttribute ->
                    root.addAttributeValue(varMap,
                                           valueForAttribute));
        return true;
    }

    var goodValues = new LinkedList<Integer>();

    // |current| is an inner node. Recurse over children.
    for (Map.Entry<Integer, Map<RelationInstance, Range>> valueAndRanges :
            valuesAndTheirRanges.entrySet().stream()
                    .sorted(Map.Entry.comparingByKey())
                    .collect(Collectors.toList()))
    {
        int valueForAttribute = valueAndRanges.getKey();

        Map<Attribute, Integer> extendedVarMap = new HashMap<>(varMap);
        extendedVarMap.put(currentAttribute, valueForAttribute);

        var restrictedRanges = valueAndRanges.getValue();
        ranges.forEach(restrictedRanges::putIfAbsent);

        boolean valuesForAllChildrenExist =
                    recComputeDrepFromDatabase(children,
                                               extendedVarMap,
                                               restrictedRanges);

        if (valuesForAllChildrenExist)
        {
            goodValues.add(valueForAttribute);
        }
    }

    if (!goodValues.isEmpty())
    {
        goodValues.forEach(value -> root.addAttributeValue(varMap, value));
        return true;
    }

    // found no good values:
    return false;
}
```

```
/**
 * Constructs a cover over a d-representation
 * @param dTree path of nodes with multimaps fully constructed
 */
public static Cover<DTreeNode> computeCoverFromDrep(List<DTreeNode> dTree,
                                                    boolean shouldPrint)
{
    Cover<DTreeNode> result = new Cover<>("Cover", dTree);

    // If d-tree is a forest:
    if (dTree.size() > 1)
    {
        Stream<Cover<DTreeNode>> coversOfChildren = dTree.stream()
                .map(child -> computeCoverFromDrep(List.of(child),
                                                   shouldPrint));

        return JoinAlgorithms.coverJoinOverDisjointSchemas(coversOfChildren);
    }

    DTreeNode root = dTree.get(0);

    List<DTreeNode> dTreeTraversal = root.preOrderTraversal().
            collect(Collectors.toList());

    for (DTreeNode node : dTreeTraversal)
    {
        Attribute currentAttribute = node.getAttribute();

        if (shouldPrint)
        {
            System.out.println("\nAt attribute: " + currentAttribute);
        }

        for (Map<Attribute, Integer> key : node.getAllKeys()
                                               .stream()
                                               .sorted(new keyComparator())
                                               .collect(Collectors.toList()))
        {
            if (shouldPrint)
            {
                System.out.println("At key: " + key);
            }

            for (var nextFreshValue = node.nextUntouchedValue(key);
                 nextFreshValue.isPresent();
                 nextFreshValue = node.nextUntouchedValue(key))
            {
                // We will build the tuple from 3 parts:
                // ancestors + currentAttribute + nextPathsFromChildren
                HashMap<Attribute, Integer> tuple;

                if (node.isRoot())
                {
                    // Do not extend the null key
```

```
        tuple = new HashMap<>();
    }
    else
    {
        Map<Attribute, Integer> fullPrefixForKey =
                node.lastPrefixForKeyValues(key);

        tuple = new HashMap<>(fullPrefixForKey);

        if (shouldPrint)
        {
            System.out.println("Ancestors: " + tuple);
        }

        List<DTreeNode> ancestors = node.ancestors()
                .collect(Collectors.toList());

        DTreeNode current = node;

        Map<Attribute, Integer> valuesForRelativesNotAncestors =
                new HashMap<>();

        // INV: current.parent = ancestor
        for (DTreeNode ancestor : ancestors)
        {
            List<DTreeNode> siblingsOfCurrent =
                    new LinkedList<>(ancestor.getChildren());
            siblingsOfCurrent.remove(current);

            Map<Attribute, Integer> pathsFromSiblingsOfCurrent =
                    siblingsOfCurrent.stream()
                            .flatMap(sibling ->
                                    sibling.nextPath(tuple)
                                            .entrySet()
                                            .stream())
                            .collect(Collectors.toMap(
                                    Map.Entry::getKey,
                                    Map.Entry::getValue));

            valuesForRelativesNotAncestors.putAll(
                    pathsFromSiblingsOfCurrent);

            current = ancestor;
        }

        tuple.putAll(valuesForRelativesNotAncestors);

        if (shouldPrint)
        {
            System.out.println("Paths from other relatives: "
                    + valuesForRelativesNotAncestors);
        }

    }
```

```
                tuple.put(currentAttribute, nextFreshValue.get());

                if (shouldPrint)
                {
                    System.out.println("=|=|=|=" + currentAttribute
                                       + "->" + nextFreshValue.get());
                }


                List<DTreeNode> children = node.getChildren();

                if (children.size() > 0)
                {
                    Map<Attribute, Integer> nextPathsFromChildren =
                            children.stream()
                                    .flatMap(child -> child.nextPath(tuple)
                                                           .entrySet()
                                                           .stream())
                                    .collect(Collectors.toMap(
                                                Map.Entry::getKey,
                                                Map.Entry::getValue));
                    tuple.putAll(nextPathsFromChildren);

                    if (shouldPrint)
                    {
                        System.out.println("Paths from children: "
                                           + nextPathsFromChildren);
                    }
                }
                  result.addTuple(tuple);
              }
        }
    }

    return result;
}



private static class keyComparator
        implements Comparator<Map<Attribute, Integer>>
{
    @Override
    public int compare(Map<Attribute, Integer> map1,
                       Map<Attribute, Integer> map2)
    {
        if (!map1.keySet().equals(map2.keySet()))
        {
            throw new RuntimeException(
                    "Cannot compare keys with different attribute sets");
        }
```

```
            List<Map.Entry<Attribute, Integer>> sortedEntries1 =
                    map1.entrySet().stream()
                                .sorted(Map.Entry.comparingByKey())
                                .collect(Collectors.toList());


            for (Map.Entry<Attribute, Integer> entry1 : sortedEntries1)
            {
                Attribute attribute1 = entry1.getKey();
                Integer value1 = entry1.getValue();
                Integer value2 = map2.get(attribute1);

                if (value1 < value2)
                    return -1;
                if (value1 > value2)
                    return 1;
            }
            return 0;
        }
    }


    public static class ListComparator implements Comparator<List<Integer>>
    {
        @Override
        public int compare(List<Integer> list1, List<Integer> list2)
        {
            if (list1.size() < list2.size())
                return -1;
            else if (list1.size() > list2.size())
                return 1;

            int index2 = 0;
            for (Integer value1 : list1)
            {
                Integer value2 = list2.get(index2++);

                if (value1 < value2)
                    return -1;
                else if (value1 > value2)
                    return 1;
            }

            return 0;
        }
    }
}
```

Bibliography

[1] S. Abiteboul, R. Hull, and V. Vianu. Foundations of Databases. Addison-Wesley, 1995.

[2] D. Olteanu, and J. Závodný. Factorised Representations of Query Results. ACM Trans. Datab. Syst. 40, 1, Article 2 (March 2015), 44 pages.

[3] D. Olteanu, and M. Schleich. Factorized Databases. SIGMOD Record, 45(2):5-16, 2016.

[4] A. Kara, and D. Olteanu. Covers of Query Results. ICDT 2018: 16:1-16:22.

[5] M. Schleich, D. Olteanu, and R. Ciucanu. Learning Linear Regression Models over Factorized Joins. In SIGMOD, pages 3-18, 2016.

[6] N. Bakibayev, T. Kocisk, D. Olteanu, and J. Zvodny. Aggregation and Ordering in Factorised Databases. PVLDB, 6(14):1990-2001, 2013.

[7] N. Bakibayev, D. Olteanu, and J. Zvodny. FDB: A query engine for factorised relational databases. Proceedings of the VLDB Endowment 5, 11, 12321243, 2012.

[8] J. Shute and et al. F1: A distributed SQL database that scales. Proceedings of the VLDB Endowment 6, 11, 1068-1079, 2013.

[9] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. J. Comput. Syst. Sci. 64, 3 (2002), 579-627.

[10] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In FOCS, pages 739-748. IEEE Computer Society, 2008.

[11] M. A. Khamis, H. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. AC/DC: In-Database Learning Thunderstruck. arXiv report 1803.07480, March 2018.

[12] M. A. Khamis, H. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. In-Database Learning with Sparse Tensors. To appear in ACM Principles of Database Systems (PODS), Houston, June 2018.

[13] X. Feng, A. Kumar, B. Recht, and C. Re. Towards a unified architecture for in-RDBMS analytics. In SIGMOD. 325336, 2012.

[14] J. M. Hellerstein and et al. The MADlib Analytics Library or MAD Skills, the SQL. PVLDB 5, 12 (2012), 17001711.

[15] D. Marx. Approximating fractional hypertree width. ACM Trans. Alg., 6(2):29:1-29:17, 2010.

[16] M. Grohe and D. Marx. Constraint solving via fractional edge covers. ACM Trans. Alg., 11(1):4, 2014.

[17] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. J. ACM, 30(3):479513, 1983.

[18] R.M. Karp. Reducibility among combinatorial problems. R.E. Miller, J.W. Thatcher (Eds.), Complexity of Computer Computations, Plenum Press, New York (1972).

[19] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In Proceedings of the 17th International Conference on Database Theory. ICDT 14. 96106, 2014.

[20] H. Q. Ngo, C. Re, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. SIGMOD Record, 42(4):5-16, 2013.

[21] M. Yannakakis. Algorithms for acyclic database schemes. In Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings, pages 82-94, 1981.

[22] M. De Domenico, A. Lima, P. Mougel and M. Musolesi. The Anatomy of a Scientific Rumor. (Nature Open Access) Scientific Reports 3, 2980, 2013.