Factorized Databases: A Knowledge Compilation Perspective

Dan Olteanu
Department of Computer Science
University of Oxford

Abstract

This paper overviews recent work on compilation of relational queries into lossless factorized representations. The primary motivation for this compilation is to avoid redundancy in the representation of query results and speed up their computation and subsequent analytics.

Introduction

The query evaluation problem is fundamental to database systems: Given a query and a relational database, the task is to compute the result of the query on the given database. The key observation underlying this work is that relations representing query results may entail a large degree of redundancy in both representation and computation.

In this paper, we advocate a knowledge compilation approach for the query evaluation problem that reduces this redundancy. The compilation target is the language of factorized databases, which are relational algebra expressions built using Cartesian product, union, and singleton relations with one tuple and one attribute. In contrast to the standard tabular representation of relations, factorized representations allow for nesting of product and union that exploits the commutativity of these two operators and the distributivity of product over union. We consider factorized relations whose nesting structures are given by orders over its attributes. For relational queries[1], these orders are over the query variables. The grounding of a variable order over a database is a factorized, indeed compact, representation of the query result that we call the factorized query.

Using knowledge compilation nomenclature, the factorized queries can be classified as d-DOMDDs, i.e., deterministic Decomposable Ordered Multi-valued Decision Diagrams.

We next introduce factorized queries and discuss their classification as d-DOMDDs and some of their properties: uniqueness, succinctness, compilation aspects, constant-delay model enumeration, and linear-time aggregates used for model counting and building regression models.

[1]For simplicity, we only discuss join queries, i.e., conjunctive queries without free variables, though the framework has been generalized to arbitrary conjunctive queries with free variables (Olteanu and Závodný 2015).

Factorized Queries by Example

Factorized databases form a representation system for relational data that exploits laws of relational algebra, such as the distributivity of the Cartesian product over union, to reduce data and computation redundancy in query processing.

Example 1 To start with a simple example, consider a relation R over schema (A, B) that consists of a tuple for each combination of values $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$. Assuming the notation $\langle A : a\rangle$ for a singleton relation over schema (A) and with one tuple with value a, the relation R can be expressed in relational algebra as $\bigcup_{1\leq i,j\leq n}\langle A : a_i\rangle \times \langle B : b_j\rangle$. A possible factorization of R is a product of two smaller relations: $R = R_A \times R_B$, where $R_A = \bigcup_{1\leq i\leq n}\langle A : a_i\rangle$ and $R_B = \bigcup_{1\leq j\leq n}\langle B : b_j\rangle$. This factorization can naturally benefit aggregates. To count the tuples in R, we take the product of the counts of tuples in R_A and R_B. To sum over all A-values in R, we multiply the sum of all A-values in R_A with the count of tuples in R_B.

We can factorize queries as well: Given a database $\mathbf{D}$ and a query Q, the query result $Q(\mathbf{D})$ exhibits lots of (data and computation) redundancy that can be reduced by factorization. Intuitively, a join of two relations is by definition a union of products of smaller relations: For every join value, several tuples from one relation can be paired with several tuples from the other relation. The factorized query avoids the materialization of these products whenever possible.

Example 2 Figure 1(a) depicts a database consisting of three relations along with their natural join: The relation House records house prices and living areas (in squared meters) within locations given by zipcodes; TaxBand relates city/state tax bands with house living areas; Shops list shops with zipcode and opening hours.

The join result exhibits a high degree of redundancy. The value z_1 occurs in 24 tuples, each value h_1 to h_3 occurs in eight tuples and they are paired with the same combinations of values for the other attributes. Since z_1 is paired in relation House with p_1 to p_3 and in relation Shops with h_1 to h_3, all combinations (indeed, the Cartesian product) of the former and the latter values occur in the join result. We can represent this local product symbolically instead of eagerly materializing it. If we systematically apply this observation, we obtain an equivalent factorized representation of the en-

Shops

Z	H
z_1	h_1
z_1	h_2
z_1	h_3
z_2	h_4

House

Z	S	P
z_1	s_1	p_1
z_1	s_1	p_2
z_1	s_2	p_3
z_2	s_2	p_4

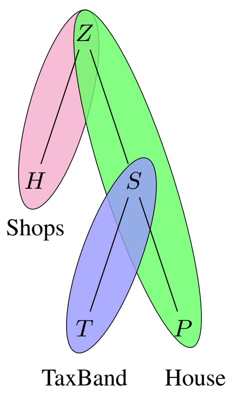
TaxBand

S	T
s_1	t_1
s_1	t_2
s_1	t_3
s_2	t_4
s_2	t_5

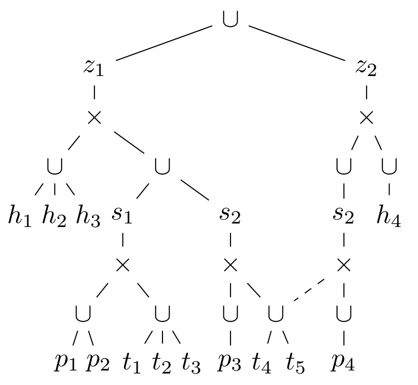
Shops ⋈ House ⋈ TaxBand

Z	H	S	P	T
z_1	h_1	s_1	p_1	t_1
z_1	h_1	s_1	p_1	t_2
z_1	h_1	s_1	p_1	t_3
z_1	h_1	s_1	p_2	t_1
z_1	h_1	s_1	p_2	t_2
z_1	h_1	s_1	p_2	t_3
z_1	h_1	s_2	p_3	t_4
z_1	h_1	s_2	p_3	t_5
....				
the above for h_2 and h_3				
....				
z_2	h_4	s_2	p_4	t_4
z_2	h_4	s_2	p_4	t_5

(a) The three relations of database $\mathbf{D}$ and natural join $Q(\mathbf{D})$.



(b) Variable order Δ.



(c) Factorized query $\Delta(\mathbf{D})$.

Figure 1: (a) Database $\mathbf{D}$ with relations House(Zipcode, Sqm, Price), TaxBand(Sqm, Tax), Shops(Zipcode, Hours), where the attribute names are abbreviated; (b) Variable order Δ for the natural join query that is the nesting structure of the factorized query; (c) Factorized query $\Delta(\mathbf{D})$ that is the grounding of the variable order Δ over the database $\mathbf{D}$.

tire join result that is much more compact than its flat, tabular representation. The factorization of the first three tuples in the join result would then be:

$$\phi = \langle Z : z_1 \rangle \times \langle H : h_1 \rangle \times \langle S : s_1 \rangle \times \langle P : p_1 \rangle \times \\ (\langle T : t_1 \rangle \cup \langle T : t_2 \rangle \cup \langle T : t_3 \rangle).$$

Figure 1(c) shows the complete factorized query (we dropped the attribute names and brackets from singletons as they are clear from context). Each tuple in the query result is represented once in the factorization and can be constructed by following one branch of every union and all branches of a product. To count the number of represented tuples, we take the sum (product) of the counts of children for each union (respectively, product) and count each singleton as one.

The factorized join in Figure 1(c) has the nesting structure depicted in Figure 1(b): It is a union of Z-singletons occurring in the join of Shops and House on Z. For each Z-singleton z, we represent separately the union of H-singletons paired with z in Shops and the union of S-singletons paired with z in House and with T-singletons in TaxBand. That is, given z, the H-singletons are independent of the S-singletons and can be stored separately; this is where the factorization saves computation and space as it avoids an explicit enumeration of all combinations of H-singletons with S-singletons for a given z. Furthermore, under each S-singleton, there is a union of T-singletons and a union of P-singletons. This nesting structure thus represents a partial order of the query variables and is captured by a tree with variables at nodes.

We can further compress the factorization by caching repeated expressions. For this, we exploit the query structure to understand what can be cached *regardless of the database content*. In our example, the union of T-singletons for a given S-singleton in TaxBand is the same regardless of how many times s occurs paired with Z and P-singletons in House, since the variable T does not *depend* on Z *given* S. This union can be defined once and reused for every occurrence of s. For instance, we define $T_{s_2} := t_4 \cup t_5$ under s_2 and all occurrences of s_2 under Z-singletons would use a pointer ${}^{\uparrow}T_{s_2}$ instead of copying T_{s_2}.

Three Factorization Flavors

In this section, we define three flavors of factorized databases (Olteanu and Závodný 2015).

Definition 3 A *factorized representation* is a list of expressions $(D_1, \ldots, D_m)$ where D_i can contain references to D_j for $j > i$ and is referenced at least once if $i > 1$. Such expressions are relational algebra expressions over a schema Σ and of the following forms:

- $\emptyset$, representing the empty relation over Σ,
- $\langle\rangle$, representing the relation consisting of the nullary tuple, if $\Sigma = \emptyset$,
- $\langle A : a \rangle$, representing the singleton relation with one tuple (a), if $\Sigma = \{A\}$ and $a \in \text{Dom}(A)$,
- $(E_1 \cup \cdots \cup E_n)$, representing the union of the relations represented by E_i, where each E_i is an expression over Σ,
- $(E_1 \times \cdots \times E_n)$, representing the Cartesian product of the relations represented by E_i, where each E_i is an expression over schema Σ_i such that Σ is the disjoint union of all Σ_i.
- a reference ${}^{\uparrow}E$ to an expression E over Σ.

We abuse notation and use interchangeably the singleton $\langle A : a \rangle$ and its value a.

The factorized representations of Definition 3 are called *d-representations*. The d-representations without definitions are called *f-representations*. For any d-representation D

consisting of expressions $\{D_1, \ldots, D_n\}$, we can start with the root expression D_1 and repeatedly replace the references ${}^{\uparrow}D_j$ by the expressions D_j until we obtain a single expression without references, which is an f-representation. The *flat representations* are f-representations where in each product $E_1 \times \cdots \times E_n$ all but at most one expression E_i are singletons. Although products of unions are not allowed in flat representations, they can be rewritten into (potentially exponentially larger) unions of permissible products. Flat representations are trie representations of relations. They are our proxy for the standard tabular representation of relations.

Variable Orders

We now turn to queries and variable orders and introduce necessary vocabulary. The size $|Q|$ of a query Q is the number n of its relation symbols. For a variable A, $rel(A)$ denotes the set of relation symbols containing A. For a set X of variables and a query Q over relation symbols $R_1, \ldots, R_n$, the X*-restriction* of Q is a join query Q_X over the relation symbols $R_1^X, \ldots, R_n^X$ restricted to variables in X.

Given a query Q, two variables A and B are *conditionally independent given* a set of variables C, if for any database $\mathbf{D}$, A's assignments do not constrain B's assignments given assignments for C in $\mathbf{D}$; otherwise, A and B are dependent. Conditional independence captures multivalued dependencies. For instance, in the join query $R_1(A, B), R_2(A, C)$, the variables B and C are independent given variable A, whereas A and B are dependent on each other as dictated by relation R_1; if A and B were independent (conditioned on the empty set of variables), then R_1 would be equal to the Cartesian product of its projections on A and B.

Definition 4 Given a join query Q, a *variable order* Δ for Q is a rooted forest with one node per variable in Q. For a variable A, let $key(A)$ be the set of ancestors of A in Δ and on which the variables in the subtree rooted at A may depend. Then, Δ satisfies the following constraints:

- The variables of each relation symbol in Q lie along the same root-to-leaf path.
- For any child B of a node A, $key(B) \subseteq key(A) \cup \{A\}$.

Variable orders serve three purposes. (1) They define the nesting structure of the factorized queries. (2) They guide the grounding process that computes the factorized query. (3) They define asymptotic size bounds for factorized queries and the time complexity to compute them.

The conditional independence of variables is modeled in a variable order by branching: Two variables A and B on different branches in a variable order Δ are conditionally independent given their common ancestors. The first constraint in Definition 4 states that all variables of a relation symbol are dependent so they cannot lie on different paths in Δ since that would mean they are independent. The second constraint captures the dependency between the variables under A and those above A: If we let $desc(A)$ be the set of variables under A, then the functional dependency $key(A) \rightarrow desc(A) \cup \{A\}$ holds. The keys of A are thus its ancestors on which A and its descendants depend.

Example 5 The keys of each variable in the variable order in Figure 1(b) are its ancestors except for $key(T) = \{S\}$.

Each of the three factorization flavors corresponds to a restriction of variable orders.

The *d-trees* are the variable orders from Definition 4 and represent the nesting structures of d-representations of query results. There is a one-to-one mapping between d-trees and hypertree decompositions of the join hypergraph.

In case not all ancestors of a variable A are in $key(A)$ in a variable order Δ, then in a d-representation over Δ the same factorization fragments rooted at A-values may be repeated for every tuple of values for variables not in $key(A)$. Here is where definitions come in handy: We give names to such factorization fragments and refer to them by their names instead of repeatedly copying them.

The *f-trees* are d-trees where for each variable A, all of its ancestors are in its key. The f-trees are the nesting structures of f-representations. In contrast to d-trees, definitions cannot save repetitions of factorization fragments in f-representations since a tuple t of values for the ancestor variables of A functionally determines the factorization fragment rooted at an A-value and in general a different fragment may occur under each distinct tuple t.

The *f-paths*, which are the variable orders for flat representations, are f-trees restricted to (forests of) paths.

Size Measures and Relative Succinctness

There may be several possible variable orders for Q and they define factorized representations of different sizes. Size measures for factorized representations are defined on the query hypergraph: For Q, the hypergraph $H(Q) = (V, E)$ has one node in the set V per query variable in Q and one (hyper)edge in the set E per relation in Q. Figure 2(a) depicts the hypergraph of the triangle query.

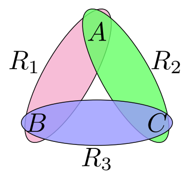
An edge cover is a subset of the edges of $H(Q)$ such that each node appears in at least one edge. Edge cover can be formulated as an integer programming problem by assigning to each edge R_i a weight x_{R_i} that can be 1 if R_i is part of the cover and 0 otherwise. The size of an edge cover upper bounds the size of the query result, since the Cartesian product of the relations in the cover includes the query result:

$$|Q(\mathbf{D})| \leq |R_1|^{x_{R_1}} \cdot \ldots \cdot |R_n|^{x_{R_n}} \leq N^{\sum_{i=1}^{n} x_{R_i}}.$$

By minimizing the size of the edge cover, we can obtain a more accurate upper bound on the size of the query result. This bound becomes tight for fractional weights (Atserias, Grohe, and Marx 2008). Minimizing the sum of the weights now becomes the objective of a linear program.

Definition 6 (Atserias, Grohe, and Marx 2008) Given a join query Q over a database $\mathbf{D} = (R_1, \ldots, R_n)$, the *fractional edge cover number* $\rho^*(Q)$ is the cost of an optimal solution to the linear program with variables $\{x_{R_i}\}_{i=1}^{n}$:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{n} x_{R_i} \\ \text{subject to} \quad & \sum_{R_i \in rel(A)} x_{R_i} \geq 1 \quad \text{for each query variable } A \\ & x_{R_i} \geq 0 \qquad \text{for each } 1 \leq i \leq n. \end{aligned}$$



(a) Hypergraph.

$$\begin{array}{rlllll} A: & x_{R_1} & + \; x_{R_2} & & \geq 1 \\ B: & x_{R_1} & & + \; x_{R_3} & \geq 1 \\ C: & & x_{R_2} & + \; x_{R_3} & \geq 1 \end{array}$$

(b) Linear program excerpt.

Figure 2: Hypergraph for the triangle query $Q_\triangleleft$ and the inequalities for query variables in the linear program for computing the tight size bound on the query result.

Figure 2(b) gives the sum-inequalities in the linear program for the fractional edge cover of the triangle query $Q_\triangleleft = R_1(A,B), R_2(A,C), R_3(B,C)$. An optimal solution is $\rho^*(Q_\triangleleft) = 3/2$ with $x_{R_1} = x_{R_2} = x_{R_3} = 1/2$. Consequently, the result of the triangle query has $O(N^{3/2})$ tuples. Furthermore, there exist classes of databases for which its size is at least $\Omega(N^{3/2})$. While the size upper bound is given in Definition 6 as a function of the query structure, the linear program may also incorporate cardinality constraints, e.g., the sizes of relations and their projections.

The fractional edge cover number is our measure for sizes of flat representations of query results.

The *f-tree width*, denoted by $s(Q)$, is the fractional edge cover number of a subquery of Q. For an f-representation over an f-tree Δ of a join query Q, the number s_A of values of a variable A is dependent on the number of possible tuples of values of its ancestors, whose set is $key(A)$, and is independent of the number of values for variables that are not on the same branch. A tight bound on s_A is then given by the fractional edge cover number of the join query that is a $(key(A) \cup \{A\})$-restriction of Q. Then, an upper bound on the size of the f-representation over Δ is the maximum over all variables in Δ of the number of values of A:

$$s(\Delta) = \max\{\rho^*(Q_{key(A)\cup\{A\}}) | A \text{ is variable in } \Delta\}$$

The f-tree width $s(Q)$ is then the minimum over all possible f-trees of the previous upper bound:

$$s(Q) = \min\{s(\Delta) | \Delta \text{ is an f-tree of } Q\}$$

The *d-tree width* $s^\uparrow(Q)$ is defined similarly to $s(Q)$, with the difference that the key of a variable may not be the set of all ancestors as for f-trees. In other words, we iterate over d-trees instead of only over their strict subset of f-trees:

$$s^\uparrow(Q) = \min\{s(\Delta) | \Delta \text{ is a d-tree of } Q\}$$

The d-tree width is equal to the fractional hypertree width of the join query (Olteanu and Závodný 2015), which is fundamental to problem tractability with applications spanning constraint satisfaction, databases, matrix operations, probabilistic graphical models, and logic (Ngo, Khamis, and Rudra 2015).

We know that $1 \leq s^\uparrow(Q) \leq s(Q) \leq \rho^*(Q) \leq |Q|$. The gap between $s(Q)$ and $\rho^*(Q)$ can be as large as $|Q|$ (e.g., for hierarchical queries), whereas the gap between $s^\uparrow(Q)$ and $s(Q)$ can be as large as $\log |Q|$, e.g., for path queries. Clique queries, e.g., triangles, are the pathological cases for which factorizations bring no asymptotic saving.

Proposition 7 (Atserias, Grohe, and Marx 2008; Olteanu and Závodný 2015) Given a join query Q, for any database $\mathbf{D}$, the join result $Q(\mathbf{D})$ admits

- a flat representation over f-paths of size $O(|\mathbf{D}|^{\rho^*(Q)})$;
- an f-representation over f-trees of size $O(|\mathbf{D}|^{s(Q)})$;
- a d-representation over d-trees of size $O(|\mathbf{D}|^{s^\uparrow(Q)})$.

There are classes of databases for which the size bounds in Proposition 7 are asymptotically tight. Furthermore, there are algorithms to compute the join result in each of the three representations in worst-case optimal time, i.e., the computation time is the same as the size bound modulo log factors in the size of the input relations (Ngo et al. 2012; Olteanu and Závodný 2015).

Example 8 A trivial upper bound for the flat representation of the natural join Q in our example is the product of the sizes of the three relations, so $O(|\mathbf{D}|^3)$ assuming each relation has size $|\mathbf{D}|$. We construct a matching lower bound for a class of databases where the Z and S values are the same across all tuples and the attributes H, T, and P have as many distinct values as $|\mathbf{D}|$. The join becomes a product of the three sets of values for H, T, and P. The fractional edge cover number $\rho^*(Q)$ is thus three.

For an f-representation, the above lower bound argument yields a size $|\mathbf{D}|^2$. The factorization for the branch $Z - H$ in the f-tree has size at most linear in $|\mathbf{D}|$, since for a Z-singleton we list the union of its H-singletons and their overall number is bounded by the number of singletons in Shops. The factorization for this branch is independent of the branch $Z - S$, since the singletons for H and S are represented independently of each other. The number of singletons for S and P is bounded by the number of singletons in House. The number of T-singletons can however be quadratic in $|\mathbf{D}|$, e.g., when one S-singleton is paired with $|\mathbf{D}|$ T-singletons in TaxBand and with $|\mathbf{D}|$ Z-singletons in House. The f-tree width $s(Q)$ is thus two.

For a d-representation, the construction used to attain the quadratic lower bound in the previous case does not work anymore: We cache the union of T-singletons for each S-singleton and reuse it under every Z-singleton. This means that the d-tree width is one. This is not surprising since Q is acyclic and its hypertree width is one.

Knowledge Compilation Perspective

Factorized queries can be presented as a special kind of an existing knowledge compilation formalism. Using appropriate nomenclature, they can be classified as d-DOMDDs, i.e., deterministic Decomposable Ordered Multi-valued Decision Diagrams. They are specifically tailored at representing relational data over a given schema and variable orders. As such, they are unique in the space of existing formalisms.

Multi-valued Decision Diagram

A factorized query is a multi-valued decision diagram: Each union node has possibly many children representing distinct values for the same attribute. Such union nodes capture the many possible values of a query variable corresponding to an attribute in the query result. For instance, the top node in Figure 1(c) is a union of two values z_1 and z_2 for the attribute Z, whereas the node under z_1 is a union of three values h_1, h_2, and h_3 for the attribute H.

Caching via Definitions

Factorized queries need not be trees, but diagrams. This is on a par with the difference between decision trees and decision diagrams, where common suffixes are cached across distinct branches. Caching is achieved via definitions and the difference between f-representations and d-representations, as explained in the previous section: For each S-value, e.g., s_2, there is a union of T-values as dictated by the relation TaxBand, e.g., $t_4 \cup t_5$. An f-representation would have a copy of the union of T-values for each occurrence of its S-value. As depicted in Figure 1(c), in a d-representation we cache the union and point to it from each occurrence of its S-value.

Ordering via Variable Orders

Factorized queries are ordered since the fragments under the children of any of its union nodes have the same variable order (nesting structure). For instance, the leftmost branch in the d-representation in Figure 1(c) has values z_1 and h_1 and their order is as in the leftmost branch in the f-tree in Figure 1(b). The branch with values z_2, s_2, p_4 follows the order of the rightmost branch.

Giving up order may bring more succinct factorizations: A free-order factorization could use different orders for the expressions under different values in a union. The difference between ordered and free-order factorized queries is on a par with the difference between OBDDs and FBDDs for Boolean formulas. As such, free-order factorizations can be more succinct than ordered ones. They would be adaptive, data-dependent, and exploit degree information in the input relations (e.g., how many S-values occur under different Z-values in relation House). For free-order flat representations, there is seminal work that gives the most general width notions known to date, called adaptive width and submodular width (Marx 2013). Our notions of f-tree width and d-tree width can be refined by using submodular width in place of the fractional edge cover number.

Determinism

Factorized queries are deterministic representations in the sense that each model (satisfying assignment of the query variables) is represented exactly once. This is ensured by two constraints: (1) Each union only has distinct values; (2) the expressions in a product represent relations over disjoint sets of attributes. Determinism is key for a host of desirable properties, including linear-time aggregates such as model counting and constant-delay model enumeration.

Decomposability

Factorized queries use decompositions to exploit multi-valued dependencies in the query results for succinct representation. For instance, the choices of opening hours for shops and the choices of house square meters or price are independent *given* the zipcode in our example. This independence is expressed using product nodes. The effect of decomposability on succinctness is great: The gap between flat representations, which are not decomposable, and f-representations, which are decomposable, can be exponential (as discussed before Proposition 7).

Properties of Factorized Queries

The factorized queries enjoy properties that make them desirable for a range of applications, as highlighted next.

Uniqueness

Given a database $\mathbf{D}$ and a variable order Δ of a query, there is one grounding of Δ over $\mathbf{D}$ modulo commutativity of product and union, which represents the query result.

Worst-case Optimal Grounding

Proposition 7 gives asymptotically tight size bounds for factorized queries within each of our representation system. Factorized queries over variable orders can be computed within time bounded by their sizes.

Constant-delay Model Enumeration

The models represented by a factorized query are the tuples in the (equivalent) relation R representing the query result. Constant-delay enumeration means that the time and extra space needed to list the first model and the time between listing two consecutive models are constant (more precisely, the delay is linear in the number arity of R, but this is constant under the usual data complexity assumption). An enumeration with constant delay is desirable since this holds for enumerating tuples from the (possibly exponentially larger) relation R. In general, given a query and a database, asking whether a given tuple is in the query result is NP-hard. Tuple enumeration remains constant-delay for acyclic queries (Bagan, Durand, and Grandjean 2007), in which case the input database together with the query already serve as a compact representation of the query result. Constant-delay model enumeration can be in *interesting* orders as long as they agree with a topological order of the variable order of the factorization (Bakibayev et al. 2013).

Linear-time Model Counting

Counting the models represented by a factorized query can be done in one pass, since factorized queries are deterministic: Turn each product node into multiplication, each union node into summation, and each singleton into value 1. The factorized query in our example encodes 26 models.

Support for Subsequent Processing

Factorized queries are materialized views that are computed once to support subsequent processing. SQL queries can be evaluated directly on them (Bakibayev, Olteanu, and Závodný 2012). A wide palette of aggregates can be processed in linear time over factorized representations: standard SQL-like aggregates with group-by clauses (Bakibayev et al. 2013) and gradient aggregates used for learning regression models (Schleich, Olteanu, and Ciucanu 2016).

Example 9 In the absence of a group-by clause, the standard aggregates count, sum, avg, min, and max can be evaluated in one pass over the factorization. We already exemplified counting. To sum the H-values in our example, we turn each union node into summation, each product node into multiplication, each H-value remains the same while each value for a different attribute turns into value 1.

In the presence of a group-by clause, the factorization may require partial restructuring. We give in earlier work a characterization of variable orders that support one-pass aggregates (Bakibayev et al. 2013). In our example, if we would like to count the number of tuples for each Z-value (i.e., we group by Z and count all tuples within each group), we only need to employ the above counting procedure under each Z-value in the top union. This is sufficient since these Z-values are non-repeating. However, if we would like to count the tuples in the query result for each S-value, we can use two strategies. The first strategy is to restructure the variable order such that the query variable S becomes root and then count the tuples represented under each S-value. The second strategy does not require restructuring. It counts the tuples containing each occurrence of an S-value and adds the counts for the same S-value.

APPLY Function for Factorizations

Arbitrary join queries can be computed by starting with flat representations of relations and incrementally combining these representations into d-representations using a multiway join operator, which takes time linear in the sum of sizes of the input factorizations (Ciucanu, Kirk, and Olteanu 2016). We next define the binary set operations difference $\dot{-}$, intersect $\Cap$, and union $\uplus$ on f-representations, and leave a complete treatment of the APPLY function for future work.

These operations are defined recursively on the structure of f-representations. We thus have three cases: singletons, unions of expressions, or products of expressions. By definition, they require that the input f-representations are over the same f-tree. Their key challenge is to preserve the determinism of the factorized representation.

For the intersection of two union expressions, we assume without loss of generality that the unions are arranged such that first $\min$ values of the two unions are the same. Since equal values in the union may nevertheless have disjoint subexpressions, we need to recurse. For product expressions, we also assume that expressions L_i and R_i have the same schema so we can intersect them. In case of intersecting two distinct values, we obtain $\bot$, which stands for empty intersection. After we finish the intersection, we can simplify the result in one bottom-up pass by (1) replacing all expressions that are connected with $\bot$ via product nodes and all unions that only have $\bot$ children by $\bot$ and (2) subsequently removing all these $\bot$ expressions. The intersection takes time linear in the sizes of the input f-representations.

$$\begin{tikzcd}
 & \cup \arrow[dl, dash] \arrow[dr, dash] & & \Cap & & \cup \arrow[dl, dash] \arrow[dr, dash] & & = & & \cup \arrow[dl, dash] \arrow[dr, dash] & \\
L_1 & \cdots & L_n & & R_1 & \cdots & R_m & & L_1 \Cap R_1 & \cdots & L_{\min} \Cap R_{\min}
\end{tikzcd}$$

$$a \Cap a = a \qquad\qquad a \Cap b = \bot$$

$$\begin{tikzcd}
 & l \arrow[d, dash] & & & & r \arrow[d, dash] & & & & l \Cap r \arrow[d, dash] & \\
 & \times \arrow[dl, dash] \arrow[dr, dash] & & \Cap & & \times \arrow[dl, dash] \arrow[dr, dash] & & = & & \times \arrow[dl, dash] \arrow[dr, dash] & \\
L_1 & \cdots & L_n & & R_1 & \cdots & R_n & & L_1 \Cap R_1 & \cdots & L_n \Cap R_n
\end{tikzcd}$$

The difference of two unions $U(L_1, \dots, L_n) \dot{-} U(R_1, \dots, R_m)$ takes the difference of the first $\min$ expressions with the same values and adds the remaining expressions from the first union, yielding:

$$\begin{tikzcd}
 & \cup \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] \arrow[drr, dash] & & \\
L_1 \dot{-} R_1 & \cdots L_{\min} \dot{-} R_{\min} & L_{\min+1} \cdots & L_n
\end{tikzcd}$$

The difference on values is as expected: $a \dot{-} a = \bot$, $a \dot{-} b = a$. For product expressions, we may incur a blowup since we need to take away from the first expression the intersection with the second expression and keep the rest. For this, we define $S = \{(X_1, \dots, X_n) | X_i \in \{(L_i \dot{-} R_i), (L_i \Cap R_i)\}, 1 \leq i \leq n\}\} - \{(L_1 \Cap R_1, \dots, L_n \Cap R_n)\}$. However, the size of S is exponential in the arity of the schema ($2^n - 1$ for n variables) and not in the data size, since the expressions of a product have disjoint sets of variables.

$$\begin{tikzcd}
 & l \arrow[d, dash] & & & & r \arrow[d, dash] & & & & l \dot{-} r \arrow[d, dash] & \\
 & \times \arrow[dl, dash] \arrow[dr, dash] & & \dot{-} & & \times \arrow[dl, dash] \arrow[dr, dash] & & = \bigcup_{(X_1, \dots, X_n) \in S} & & \times \arrow[dl, dash] \arrow[dr, dash] & \\
L_1 & \cdots & L_n & & R_1 & \cdots & R_n & & X_1 & \cdots & X_n
\end{tikzcd}$$

The union of two f-representations L and R can then be expressed as $L \uplus R = (L \Cap R) \cup (L \dot{-} R) \cup (R \dot{-} L)$.

Factorization: Old Friend in New Clothes

Factorizing data and computation is commonplace across Computer Science. Query hypergraphs define probabilistic graphical models (PGMs), where the hyperedges and query variables become factors and random variables, respectively. Inference is thus expressible using sum and count aggregates on factorized queries. By this connection, our framework gives new algorithms and complexity results for inference in PGMs (there are classes of instances with unbounded treewidth and $s^{\uparrow}$ one). Recent work shows how factorized computation can effectively solve a host of problems in databases, logic, CSP, coding theory, and matrix operations (Ngo, Khamis, and Rudra 2015).

The closest in spirit to our factorization framework are the d-DNNFs over vtrees, which also exploit structural decomposability of the input (Pipatsrisawat and Darwiche 2008), and the multivalued decomposable decision graphs (Koriche et al. 2015). Our framework is a special instance of these two and distinct due to its root in relational databases. Factorizations represent relations over a fixed schema and not arbitrary expressions. They draw on the separation of queries from data for defining variable orders and associated complexity results. Finally, they are specifically designed to support aggregates and queries. Related work in databases is reviewed elsewhere (Olteanu and Závodný 2015).

Acknowledgements. The author would like to thank his former and current group members who contributed to this work: Nurzhan Bakibayev, Radu Ciucanu, Tomáš Kočiský, Max Schleich, and Jakub Závodný.

References

Atserias, A.; Grohe, M.; and Marx, D. 2008. Size bounds and query plans for relational joins. In *FOCS*, 739–748.

Bagan, G.; Durand, A.; and Grandjean, E. 2007. On acyclic conjunctive queries and constant delay enumeration. In *Computer Science Logic*, 208–222.

Bakibayev, N.; Kociský, T.; Olteanu, D.; and Závodný, J. 2013. Aggregation and ordering in factorised databases. *PVLDB* 6(14):1990–2001.

Bakibayev, N.; Olteanu, D.; and Závodný, J. 2012. FDB: A query engine for factorised relational databases. *PVLDB* 5(11):1232–1243.

Ciucanu, R.; Kirk, J.; and Olteanu, D. 2016. Worst-case optimal join at a time. In *(under submission)*.

Koriche, F.; Lagniez, J.; Marquis, P.; and Thomas, S. 2015. Compiling constraint networks into multivalued decomposable decision graphs. In *IJCAI*, 332–338.

Marx, D. 2013. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *J. ACM* 60(6):42.

Ngo, H. Q.; Porat, E.; Ré, C.; and Rudra, A. 2012. Worst-case optimal join algorithms. In *PODS*, 37–48.

Ngo, H. Q.; Khamis, M. A.; and Rudra, A. 2015. FAQ: Questions Asked Frequently, CoRR:1504.04044.

Olteanu, D., and Závodný, J. 2015. Size bounds for factorised representations of query results. *TODS* 40(1):2.

Pipatsrisawat, K., and Darwiche, A. 2008. New compilation languages based on structured decomposability. In *AAAI*, 517–522.

Schleich, M.; Olteanu, D.; and Ciucanu, R. 2016. Learning linear regression models over factorized joins. In *SIGMOD*.

Update to the published version

We corrected the worst-case optimality statements in Proposition 7 and added clarification in the subsequent paragraph.