# Factorized Databases

http://www.cs.ox.ac.uk/projects/FDB/

Dan Olteanu Maximilian Schleich

Department of Computer Science, University of Oxford

## ABSTRACT

This paper overviews factorized databases and their application to machine learning. The key observation underlying this work is that state-of-the-art relational query processing entails a high degree of redundancy in the computation and representation of query results. This redundancy can be avoided and is not necessary for subsequent analytics such as learning regression models.

## 1. INTRODUCTION

Succinct data representations have been developed across many fields including computer science, statistics, applied mathematics, and signal processing. Such representations are employed for instance for storing and transmitting otherwise large amounts of data, and for speeding up data analysis [22].

In this paper we overview recent developments on factorized databases, which are succinct lossless representations of relational data. They exploit laws of relational algebra, in particular the distributivity of the Cartesian product over union that underlies algebraic factorization, and data and computation sharing to reduce redundancy in the representation and computation of query results. The relationship between a flat, tabular representation of a relation as a set of tuples and an equivalent factorized representation is on a par with the relationship between logic functions in disjunctive normal form and their equivalent circuits.

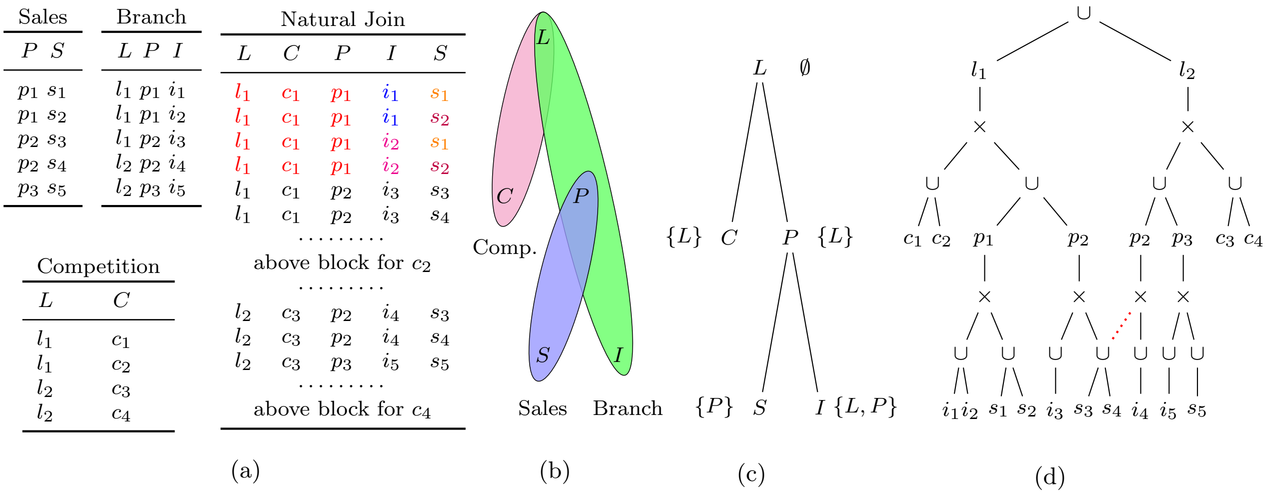
Factorized databases naturally capture existing relational decompositions proposed in the literature: lossless decompositions defined by join dependencies, as investigated in the context of normal forms in database design [1], conditional independence in Bayesian networks [18], minimal constraint networks in constraint satisfaction [7], factorizations of provenance polynomials of query results [15] used for efficient computation in probabilistic databases [12, 20], and product decompositions of relations as studied in the context of incomplete information [13].

In the following we first exemplify the benefits of factorizing query results. We then discuss factorizations for various classes of queries, quantify the succinctness gap between factorized and standard tabular representations for results of conjunctive queries and survey worst-case optimal algorithms for computing factorized representations of query results [16, 17]. We then briefly mention the case of queries with aggregates and order-by clauses [3] and discuss in more detail their application to learning regression models over factorized databases [19, 14].

## 2. A FACTORIZATION EXAMPLE

Figure 1(a) depicts three relations and their natural join. `Branch` records the location, products and daily inventory of each branch store in the chain. There are many products per location and many inventories per product. `Competition` records the competitors (e.g., the distance to competitor stores) of a store branch at a given location, with several competitors per location. `Sales` records daily sales offered by the store chain for each product.

The join result exhibits a high degree of redundancy. The value $l_1$ occurs in 12 tuples, each value $c_1$ and $c_2$ occurs in six tuples and they are paired with the same tuples of values for the other attributes. Since $l_1$ is paired in `Competition` with $c_1$ and $c_2$ and in `Branch` with $p_1$ and $p_2$, the Cartesian product of $\{c_1, c_2\}$ and $\{p_1, p_2\}$ occurs in the join result. We can represent this product symbolically as $\{c_1, c_2\} \times \{p_1, p_2\}$ instead of materializing it. If we systematically apply this observation, we obtain an equivalent *factorized representation* of the entire join result that is much more compact than its flat representation. Each tuple in the flat join result is represented once in the factorization and can be constructed by following one branch of each union and all branches of each product. The flat join result in Figure 1(a) has 90 values (18 tuples of 5 values each), while the equivalent factorized join result in Figure 1(d) only has 20 values.



| Sales | |
|---|---|
| $P$ | $S$ |
| $p_1$ | $s_1$ |
| $p_1$ | $s_2$ |
| $p_2$ | $s_3$ |
| $p_2$ | $s_4$ |
| $p_3$ | $s_5$ |

| Branch | | |
|---|---|---|
| $L$ | $P$ | $I$ |
| $l_1$ | $p_1$ | $i_1$ |
| $l_1$ | $p_1$ | $i_2$ |
| $l_1$ | $p_2$ | $i_3$ |
| $l_2$ | $p_2$ | $i_4$ |
| $l_2$ | $p_3$ | $i_5$ |

| Competition | |
|---|---|
| $L$ | $C$ |
| $l_1$ | $c_1$ |
| $l_1$ | $c_2$ |
| $l_2$ | $c_3$ |
| $l_2$ | $c_4$ |

| Natural Join | | | | |
|---|---|---|---|---|
| $L$ | $C$ | $P$ | $I$ | $S$ |
| $l_1$ | $c_1$ | $p_1$ | $i_1$ | $s_1$ |
| $l_1$ | $c_1$ | $p_1$ | $i_1$ | $s_2$ |
| $l_1$ | $c_1$ | $p_1$ | $i_2$ | $s_1$ |
| $l_1$ | $c_1$ | $p_1$ | $i_2$ | $s_2$ |
| $l_1$ | $c_1$ | $p_2$ | $i_3$ | $s_3$ |
| $l_1$ | $c_1$ | $p_2$ | $i_3$ | $s_4$ |
| ········· | | | | |
| above block for $c_2$ | | | | |
| ········· | | | | |
| $l_2$ | $c_3$ | $p_2$ | $i_4$ | $s_3$ |
| $l_2$ | $c_3$ | $p_2$ | $i_4$ | $s_4$ |
| $l_2$ | $c_3$ | $p_3$ | $i_5$ | $s_5$ |
| ········· | | | | |
| above block for $c_4$ | | | | |

**Figure 1: (a) Database with relations Branch(Location, Product, Inventory), Competition(Location, Competitor), Sales(Product, Sale), where the attribute names are abbreviated; (b) Hypergraph of the natural join of the relations; (c) Variable order $\Delta$ defining one possible nesting structure of the factorized join result given in (d). The union $s_3 \cup s_4$ is cached under the first occurrence of $p_2$ and referenced (via a dotted edge) from the second occurrence of $p_2$.**

Figure 1(c) depicts the nesting structure of our factorized join result as a partial order $\Delta$ on the query variables: The factorization is a union of $L$-values occurring in both Competitors and Branch. For each $L$-value $l$, it is a product of the union of $C$-values paired with $l$ in Competitors and of the union of $P$-values paired with $l$ in Branch and with $S$-values in Sales. That is, given $l$, the $C$-values are *independent* of the $P$-values and can be stored separately. The factorization saves computation and space as it avoids the materialization of the product of the unions of $C$-values and of $P$-values for a given $L$-value. The same applies to the unions of $S$-values and of $I$-values under each $P$-value. Further saving is brought by *caching* expressions: The union of $S$-values $S_{34} = s_3 \cup s_4$ from Sales occurs with the $P$-value $p_2$ *regardless* of which $L$-values $p_2$ is paired with in Branch, so we can store the first occurrence of $S_{34}$ and refer to it using a pointer ${}^{\uparrow}S_{34}$ from every subsequent occurrence of $p_2$. Like product factorization, caching is enabled by conditional independence: The variable $S$ is independent of its ancestor $L$ given its parent $P$. We encode it using a function *key* that maps each variable $A$ to the set of its ancestors on which $A$ and its descendants depend; this is given next to each variable in $\Delta$.

Different variable orders are possible. We seek those variable orders that fully exploit the independence among variables and lead to succinct factorizations. Branching and caching are indicators of good variable orders. The total orders have no branching and caching, so they define factorizations with no asymptotic saving over flat representations.

For our join and any database, a factorized join result can be computed in linear time (modulo a log factor in the database size). In contrast, there are databases for which the flat join result requires cubic computation time, e.g., databases with one $L$ and $P$-value and $n$ distinct $C$, $S$, and $I$-values.

SQL aggregates can be computed in one pass over the factorized join result. For instance, to compute the aggregate `sum(1)` that computes the cardinality of the join result, we interpret each data value as 1 and turn unions into sums and products into multiplication. To compute `sum(P*C)` that sums over all multiplications of products and competitors (assuming they are numbers), we turn all values except for $P$ and $C$ into 1, unions into sums, and products into multiplications (the result of $\boxed{(1{+}1)}$ in the first line is cached and reused in the second line):

$$1 \cdot (c_1 + c_2) \cdot [p_1 \cdot (1+1) \cdot (1+1) + p_2 \cdot 1 \cdot \boxed{(1{+}1)}] +$$
$$1 \cdot [p_2 \cdot \boxed{(1{+}1)} \cdot 1 + p_3 \cdot 1 \cdot 1] \cdot (c_3 + c_4).$$

Learning regression models requires the computation of a family of aggregates `sum(`$X_1$`*...*`$X_n$`)` for any tuple of (not necessarily distinct) variables $(X_1, \ldots, X_n)$, such as the above aggregate `sum(P*C)`.

We may also compute aggregates with group-by clauses. Our factorized join result supports grouping by any set of variables that sit above all others in the variable order $\Delta$, e.g., group by $\{L, C, P\}$.

# 3. QUERY FACTORIZATION

As exemplified in Section 2, factorized representations of relational data use Cartesian products to capture the independence in the data, unions to captures alternative values for an attribute, and references to capture caching.

Definition 3.1. A *factorized representation* is a list $(D_i)_{i\in[m]}$, where each $D_i$ is a relational algebra expression over a schema $\Sigma$ and has one of the following forms:

- $\emptyset$, representing the empty relation over $\Sigma$,
- $\langle\rangle$, representing the relation consisting of the nullary tuple, if $\Sigma = \emptyset$,
- $a$, representing the relation $\{(a)\}$ with one tuple having one data value $(a)$, if $\Sigma = \{A\}$ and the value $a \in \text{Dom}(A)$,
- $\bigcup_{j\in[k]} E_j$, representing the union of the relations represented by $E_j$, where each $E_j$ is an expression over $\Sigma$,
- $\times_{j\in[k]} E_j$, representing the Cartesian product of the relations represented by $E_j$, where each $E_j$ is an expression over schema $\Sigma_j$ such that $\Sigma$ is the disjoint union of all $\Sigma_j$.
- a reference ${}^{\uparrow}E$ to an expression $E$ over $\Sigma$.

The expression $D_i$ may contain references to $D_k$ for $k > i$ and is referenced at least once if $i > 1$. □

Definition 3.1 allows arbitrarily-nested factorized representations. In this paper, we focus on factorized representations of query results whose nesting structures are given by orders on query variables.

Definition 3.2. Given a join query $Q$, a variable *depends* on another variable if they occur in the same relation symbol in $Q$.

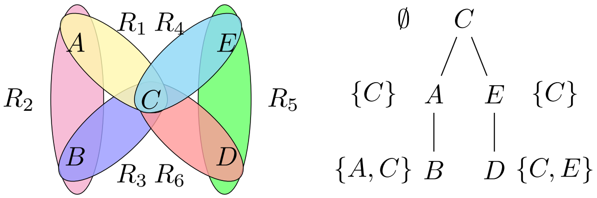
A *variable order* $\Delta$ for $Q$ is a pair $(T, key)$.

- $T$ is a rooted forest with one node per variable in $Q$ such that the variables of each relation symbol in $Q$ lie along the same root-to-leaf path in $T$.
- The function *key* maps each variable $A$ to the subset of its ancestor variables in $T$ on which the variables in the subtree rooted at $A$ depend, i.e., for every variable $B$ that is a child of a variable $A$, $key(B) \subseteq key(A) \cup \{A\}$. □

If two variables $A$ and $B$ in $Q$ depend on each other, then the choice of $A$-values may restrict the choice of $B$-values in a factorization of $Q$'s result and we need to represent explicitly their possible combinations. If they are independent, then the set of $A$-values can be represented separately from the set of $B$-values and their combinations are only expressed symbolically. The succinctness of factorized representations lies in the exploitation of conditional independence between variables. In a variable order, this is reflected in branching, i.e., a variable has several children, and caching, i.e., a variable has ancestors that are not keys.

Example 3.3. The key information in the variable order $\Delta$ in Figure 1(c) is given next to each variable. The variable $L$ is an ancestor of $S$, yet it is not in $key(S)$ since $S$ does not depend on it. We can thus cache the unions of $S$-values for a $P$-value and refer to them for each occurrence of that $P$-value. $\Delta$ also features branching: $C$ and $P$ are children of $L$, while $S$ and $I$ are children of $P$. This means that for a given $L$-value $(P)$, we can store symbolically the product of the unions of $C$-values and of $P$-values (and respectively of $S$ and $I$).

Consider now the cyclic bowtie join query over relations $R_1, \ldots, R_6$ and a variable order for it:



For each variable, its keys coincide with its ancestors, so there is no saving due to caching. There are however two branches under $C$, each of them defining a triangle query. For each $C$-value $c$ we can thus compute and store the set of triangles $\{(c, A, B) \mid R_1(A, c), R_2(A, B), R_3(B, c)\}$ for the left branch separately from the set of triangles $\{(c, E, D) \mid R_4(c, E), R_5(E, D), R_4(c, D)\}$ for the right branch. □

Among the known classes of variable orders [17, 6], we consider here the most general class called d-trees. They are another syntax for hypertree decompositions of the join hypergraph [17].

## 3.1 Succinctness and Computation Time

The construction of variable orders is guided by the joins, their selectivities, and input cardinalities. They can lead to factorizations of greatly varying sizes, where the size of a representation (flat or factorized) is defined as the number of its values. Within the class of factorizations over variable orders, we can find the worst-case optimal ones and also compute them in worst-case optimal time:

Theorem 3.4 ([2, 11, 17]). *Given a join query $Q$, for every database $\mathbf{D}$, the result $Q(\mathbf{D})$ admits*

- *a flat representation of size $O(|\mathbf{D}|^{\rho^*(Q)})$;*
- *a factorized representation of size $O(|\mathbf{D}|^{fhtw(Q)})$.*

*There are classes of databases $\mathbf{D}$ for which the above size bounds are tight.*

*There are worst-case optimal join algorithms to compute the join result in these representations.*

```
**factorize** (variable order $\Delta$, varMap, ranges$[(start_i, end_i)_{i \in [r]}]$)

**if** $(\Delta = (\Delta_j)_{j \in [k]})$ **return** $\times_{j \in [k]}$ **factorize**$(\Delta_j, \text{varMap}, \text{ranges}[(start_i, end_i)_{i \in [r]}])$;

$A = var(\Delta)$;   $E_\Delta = \emptyset$;   context $= \pi_{key(A)}(\text{varMap})$;

**if** $(key(A) \neq anc(A))$  {  ${}^{\uparrow}E_\Delta = \text{cache}_A[\text{context}]$;  **if** ${}^{\uparrow}E_\Delta \neq 0$ **return** ${}^{\uparrow}E_\Delta$;  }

**foreach** $a \in \bigcap_{i \in [r], A \in \mathsf{Schema}[R_i]} \pi_A(R_i[start_i, end_i])$ **do** {
  **foreach** $i \in [r]$ **do** find ranges $R_i[start'_i, end'_i] \subseteq R_i[start_i, end_i]$ s.t. $\pi_A(R_i[start'_i, end'_i]) = a$;
  **switch**$(\Delta)$ :
    **leaf node** $A$ :
        $E_\Delta = E_\Delta \cup a$;
    **inner node** $A(\Delta_j)_{j \in [k]}$ :
        **foreach** $j \in [k]$ **do** $E_{\Delta_j}$= **factorize**$(\Delta_j, \text{varMap} \times a, \text{ranges}[(start'_i, end'_i)_{i \in [r]}])$;
        **if** $(\forall j \in [k] : E_{\Delta_j} \neq \emptyset)$ $E_\Delta = E_\Delta \cup \big(a \times (\times_{j \in [k]} E_{\Delta_j})\big)$;
}
**if** $(key(A) \neq anc(A))$  $\text{cache}_A[\text{context}] = {}^{\uparrow}E_\Delta$;

**return** $E_\Delta$;
```

**Figure 2: Grounding a variable order $\Delta$ over a database $(R_1, \ldots, R_r)$. The parameters of the initial call are $\Delta$, an empty variable map, and the full range of tuples for each relation.**

The measures $\rho^*(Q)$ and $fhtw(Q)$ are the fractional edge cover number and the fractional hypertree width respectively. We know that

$$1 \leq fhtw(Q) \leq \rho^*(Q) \leq |Q|$$

and the gap between them can be as large as $|Q|$, which is the number of relations in $Q$. The fractional hypertree width is fundamental to problem tractability with applications spanning constraint satisfaction, databases, matrix operations, logic, and probabilistic graphical models [9].

Example 3.5. The join query in Section 2 is acyclic and has $fhtw = 1$ and $\rho^* = 3$. The bowtie query has $fhtw = 3/2$, which already holds for each of its two triangles, and $\rho^* = 3$, which is the sum of the $\rho*$ values of the two triangles. □

## 3.2 Worst-case Optimal Join Algorithms

Worst-case optimal join algorithms for flat query results have been developed only recently [11]. At their outset is the observation that the classical relation-at-a-time query plans are suboptimal since their flat intermediate results may be larger than the flat query result [2]. To attain worst-case optimality, a new breed of join algorithms has been proposed that avoids intermediate results [11]. This monolithic recipe is however an artifact of the flat representation and not necessary for optimality: Using factorized intermediate results, optimality can be achieved by join-at-a-time query plans [6]. Such plans explore breadth-first the factorized space of assignments for query variables and can compute the join result in both factorized and flat form. An equivalent depth-first exploration leads to a monolithic worst-case optimal algorithm [17].

Figure 2 gives a worst-case optimal monolithic (depth-first) algorithm that computes the grounding $E_\Delta$ of a variable order $\Delta$ over an input database. If $\Delta$ is a variable order for a join query, then $E_\Delta$ is the factorized join result. As discussed in Section 3.3, $\Delta$ may also be a variable order for conjunctive queries with group-by and order-by clauses.

In case $\Delta$ is a forest, we construct a product of the factorizations over its trees. We next discuss the case where $\Delta$ is a tree with root variable $A$.

The relations are assumed sorted on their attributes following a depth-first pre-order traversal of $\Delta$. Each call takes a range defined by start and end indices in each relation. Initially, these ranges span the entire relations. Once the root $A$ is mapped to a value $a$ in the intersection of possible $A$-values from the relations with attribute $A$, then these ranges are narrowed down to those tuples with value $a$ for $A$. We may further narrow down these ranges using mappings for variables below $A$ in $\Delta$ at higher recursion depths. Each $A$-value $a$ in this intersection is the root of a factorization fragment over $\Delta$. (Following Definition 3.1, $a$ stands for relation $\{(a)\}$.)

We first check whether the factorization we are about to compute has been already computed. If this is the case, we simply return a reference to it from cache. If not, we compute it and place its reference in the cache. The key for the cache is the context of $A$, i.e., the current mapping of the variables in $key(A)$. The current variable mappings are kept in varMap. Caching is useful when $key(A)$ is strictly contained in $anc(A)$, since this means that the factorization fragments over the variable order rooted at $A$ are repeated for every distinct combination of values for variables in $anc(A) \setminus key(A)$.

If $\Delta$ is a leaf node, then we construct a union of all mappings $a$ of $A$. If it is an inner node, then for each mapping $a$ we recurse to each child and construct a factorization that is a product of $a$ and the factorizations at children.

This algorithm defaults to LeapFrog TrieJoin [23] if $\Delta$ is a path where $key(A) = anc(A)$ for each variable $A$ in $\Delta$, i.e., when there is no branching and no sharing. The resulting factorization is a trie.

The key operation dictating the time complexity of this algorithm is the intersection of the arrays of ordered values defined by the relation ranges. This takes time linear in the size of the smallest array (modulo log factor) [23].

### 3.3 Beyond Join Queries

The above framework is immediately extensible to queries with projections [16, 17] and order-by and group-by clauses [3] by appropriately restricting the variable orders of the factorized query results.

**Projection.** If a variable is projected away, then all variables depending on it now depend on each other. Following Definition 3.2, all dependent variables need to lie along the same path in the variable order. For instance, if we project away the variable $P$ in our running example, then the variables $S$ and $I$, which used to be independent given $P$, become dependent on each other. This restricts the possible variable orders of factorizations of the query result, possibly decrease the branching factor in the variable order, and likely increases the factorization size. Variable orders and their widths can be defined for conjunctive queries in immediate analogy to the case of join queries [16, 17].

**Group-By and Order-By.** For a relation representing a query result $R$, grouping by a set $G$ of variables partitions the tuples of $R$ into groups that agree on the $G$-value. Ordering $R$ by a list $O$ of variables sorts $R$ lexicographically on the variables in the order given by $O$, where for each variable in $O$ the sorting is in ascending or descending order.

Given a factorized representation $R$ over a variable order, we can enumerate the tuples in the relation represented by $R$ *in no particular order* with *constant delay*, i.e., the time between listing two consecutive tuples is independent of the number of tuples and thus constant under data complexity.

Group-by and order-by clauses require however to enumerate the tuples *in some desired order* as given by the explicit order $O$ or by the group $G$ so that all tuples with the same $G$-value are listed consecutively and can be aggregated. Constant-delay enumeration following an order $O$ or a group $G$ is not supported by arbitrary variable orders, but by those obeying specific constraints on the variables in $G$ or $O$.

Theorem 3.6 ([3]). *Given a factorized representation of a relation $R$ over a variable order $\Delta$, a set $G$ of group-by variables, and a list $O$ of order-by variables.*

*The tuples within each $G$-group in $R$ can be enumerated with constant delay if and only if each variable of $G$ is either root in $\Delta$ or a child of another variable of $G$.*

*The tuples in $R$ can be enumerated with constant delay in sorted lexicographic order by $O$ if and only if each variable $X$ of $O$ is either root in $\Delta$ or a child of a variable appearing before $X$ in $O$.*

In other words, constant-delay enumeration of tuples within a group holds exactly when the group-by variables are above the other variables in the variable order $\Delta$. For an order-by list $O$, the condition is stronger: there is a topological order of the variables in $\Delta$ that has $O$ as prefix. Theorem 3.6 assumes that the values within each union are sorted. This is the case in the FDB system for factorized query processing [4] and the F system for factorized learning of regression models [19].

Example 3.7. The variable order $\Delta$ from Figure 1(c) supports constant-delay tuple enumeration for the following sets of group-by variables: $\{L\}$, $\{L, C\}$, $\{L, P\}$, $\{L, C, P\}$, $\{L, P, I\}$, $\{L, P, I, S\}$, $\{L, C, P, S\}$, $\{L, C, P, I\}$, and $\{L, C, P, S, I\}$; and for the lists of order-by variables (with any variable in ascending or descending order) that can be constructed from the above sets such that they are prefixes of a topological order of $\Delta$. □

There are two strategies for a factorized computation of a conjunctive query $Q$ with order-by or group-by clauses. We derive a variable order $\Delta$ with minimum width that satisfies all constraints for the joins, projection, and order-by or group-by clauses. Then, given $\Delta$ and the input database, we compute the factorized query result. Alternatively, we derive a variable order $\Delta'$ for the join of $Q$ and compute the factorized join result. We then restructure $\Delta'$ and its factorization to support projection, grouping, and ordering[3]. The second strategy may be preferred if the join is very selective so the restructuring is not expensive.

**Aggregates.** We consider SQL aggregates based on expressions in the semiring $(\mathbb{N}[\Delta], +, \cdot, 0, 1)$ of polynomials with variables from $\Delta$ and coefficients from $\mathbb{N}$ [8], e.g., sums over expressions $2 \cdot X$ and $X \cdot Y$ for variables $X$ and $Y$.

THEOREM 3.8 (GENERALIZATION OF [3, 19]). *Given a variable order $\Delta$ and a factorized representation $E$ over $\Delta$. Any SQL aggregate of the form* `sum(X)`, `min(X)`, *or* `max(X)`, *where $X$ is an expression in the semiring $(\mathbb{N}[\Delta], +, \cdot, 0, 1)$, can be computed in one pass over $E$.*

If $\Delta$ supports constant-delay enumeration for a group-by clause $G$, then Theorem 3.8 applies to SQL aggregates with group-by $G$ clause.

The algorithm in Figure 2 can be extended to compute aggregates on top of joins without the need to first materialize the factorized join result [19]: Instead of creating factorization fragments and possibly caching them, we compute (and possibly cache) the result of computing the aggregates on them and propagate these aggregates up through recursion. Since the aggregates we consider are distributive, we compute them at a variable in the variable order using the aggregates at its children.

Our earlier worst-case optimal factorization algorithm for conjunctive queries [17] coupled with one-pass aggregates and group-by clauses [3] can be recovered via the recent framework of Functional Aggregate Queries (FAQ) [9]. Both approaches are dynamic programming algorithms with the same runtime complexity. While FAQ is bottom-up, the factorization algorithm is top-down (memoized).

# 4. LEARNING REGRESSION MODELS

We show in this section how to learn polynomial regression models over factorized databases, generalizing earlier work on linear regression [19]. The core computation underlying the construction of such models concerns a set of aggregates with semiring expressions such as those from Theorem 3.8.

## 4.1 Polynomial Regression

We consider the setting where a polynomial regression model is learned over a training dataset defined by a query over a database:

$$\{(y^{(1)}, x_1^{(1)}, \ldots, x_n^{(1)}), \ldots, (y^{(m)}, x_1^{(m)}, \ldots, x_n^{(m)})\}.$$

The values $y^{(i)}$ are the *labels* and the values $x_j^{(i)}$ are the *features*. We assume that the intercept of the model is captured by one feature with value 1. All values are real numbers.

Polynomial regression models predict the value of the label based on a linear function of parameters and *feature interactions*, which are products of features. A polynomial regression model with all feature interactions up to degree $d$ is given by:

$$h_\theta(x) = \sum_{t=1}^{d} \sum_{k_1=1}^{n} \cdots \sum_{k_t=k_{t-1}}^{n} \theta_{(k_1,\ldots,k_t)} x_{k_1} \cdot \ldots \cdot x_{k_t}.$$

The case of $d = 1$ corresponds to linear regression.

For each model $h_\theta$, we define a set $\mathcal{I}$ such that each $K \in \mathcal{I}$ corresponds to the feature interaction of parameter $\theta_K$ in $h_\theta$. For example, by substituting $K$ by $(1, 3)$, the parameter $\theta_{(1,3)}$ corresponds to the interaction term $x_1 \cdot x_3$. For the model given above:

$$\mathcal{I} = \bigcup_{t \in [d]} \{(k_1, .., k_t) \mid \forall 1 \leq k_1 \leq \ldots \leq k_t \leq n\}.$$

Given the training dataset, the goal is to fit the parameters $\theta_{(k_1,\ldots,k_t)}$ of the model so as to minimize the error of an objective function. We consider the popular least squares regression objective function:

$$\mathcal{E}(\theta) = \frac{1}{2} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda R(\theta).$$

$R(\theta)$ is a regularization term used to overcome model overfitting, and $\lambda$ determines its weight in $\mathcal{E}(\theta)$. Examples of regularization terms are: $\lambda \sum_{J \in \mathcal{I}} \theta_J^2$ (Ridge); $\lambda \sum_{J \in \mathcal{I}} |\theta_J|$ (Lasso); and $\lambda_1 \sum_{J \in \mathcal{I}} |\theta_J| + \lambda_2 \sum_{J \in \mathcal{I}} \theta_J^2$ (Elastic-Net). For uniformity in our equations, we treat the label $y$ as a feature but with a predefined parameter -1.

We use *batch gradient descent* (BGD) [5] to learn the model. BGD repeatedly updates the model parameters in the direction of the gradient to decrease the error given by the objective function $\mathcal{E}(\theta)$ and to eventually converge to the optimal value:

$$\forall J \in \mathcal{I} : \theta_J := \theta_J - \alpha \frac{\delta}{\delta \theta_J} \mathcal{E}(\theta),$$

$$\frac{\delta}{\delta \theta_J} \mathcal{E}(\theta) = \sum_{i=1}^{m} h_\theta(x^{(i)}) \prod_{j \in J} x_j^{(i)} + \lambda \frac{\delta}{\delta \theta_J} R(\theta),$$

where the learning rate $\alpha$ determines the size of each convergence step. We use $j \in J$ to iterate over the elements in tuple $J$.

In standard BGD, each convergence step scans the training dataset and computes the sum aggregate, then updates the parameters, and repeats this process until convergence. This is inefficient because a large bulk of the computation is repeated across the convergence steps. A rewriting of the sum aggregate can avoid the redundant work and make it very competitive.

## 4.2 Rewriting the Update Program

BGD has two logically independent tasks: The computation of the sum aggregate and convergence of the parameters. The data-dependent part is the sum aggregate $S_J$, where $J, K = (k_1, \ldots, k_t) \in \mathcal{I}$:

$$S_J = \sum_{i=1}^{m} \left( \sum_{t=1}^{d} \sum_{k_1=1}^{n} \cdots \sum_{k_t=k_{t-1}}^{n} \theta_K \prod_{k \in K} x_k^{(i)} \right) \prod_{j \in J} x_j^{(i)}$$

We can explicate the *cofactor* of $\theta_K$ in $S_J$:

$$S_J = \sum_{t=1}^{d} \sum_{k_1=1}^{n} \cdots \sum_{k_t=k_{t-1}}^{n} \theta_K \times \text{Cofactor}[K, J]$$

$$\text{where Cofactor}[K, J] = \sum_{i=1}^{m} \prod_{k \in K} x_k^{(i)} \prod_{j \in J} x_j^{(i)}.$$

For linear regression, $t = p = 1$ and the cofactors are sums over products of two features.

Remarkably, the data-dependent computation is captured fully by the cofactors, which are completely decoupled from the parameters. Therefore, this reformulation enables us to compute cofactors once and perform parameter convergence directly on the matrix of cofactors, whose size is independent of the data size $m$. This is crucial for performance as we do not require one pass over the entire training dataset for each convergence step.

The matrix of cofactors has desirable properties:

Proposition 4.1. ([19]) *Given a query $Q$, database $\mathbf{D}$, where the query result $Q(\mathbf{D})$ has schema $\sigma = (A_i)_{i \in [n]}$. Let* Cofactor *be the cofactor matrix for learning a polynomial regression model $h_\theta$ using BGD over $Q(\mathbf{D})$.*

*The cofactor matrix has the following properties:*

*1.* Cofactor *is symmetric:*

$$\forall K, J \in \mathcal{I} : \text{Cofactor}[K, J] = \text{Cofactor}[J, K].$$

*2.* Cofactor *computation commutes with union: Given a disjoint partitioning $\mathbf{D} = \bigcup_{j \in [p]}(\mathbf{D}_j)$ and cofactors $(\text{Cofactor}_j)_{j \in [p]}$ over $(Q(\mathbf{D}_j))_{j \in [p]}$, then*

$$\forall K, J \in \mathcal{I} : \text{Cofactor}[K, J] = \sum_{j=1}^{p} \text{Cofactor}_j[K, J].$$

*3.* Cofactor *computation commutes with projection: Given a feature set $L \subseteq \sigma$ and cofactor matrix* $\text{Cofactor}_L$ *for the training dataset $\pi_L(Q(\mathbf{D}))$, then*

$$\forall K, J \in \mathcal{I}(\text{ s.t. } \forall k \in K, j \in J : A_k, A_j \in L) : \\ \text{Cofactor}_L[K, J] = \text{Cofactor}[K, J].$$

The symmetry property implies that we only need to compute one half of the cofactor matrix.

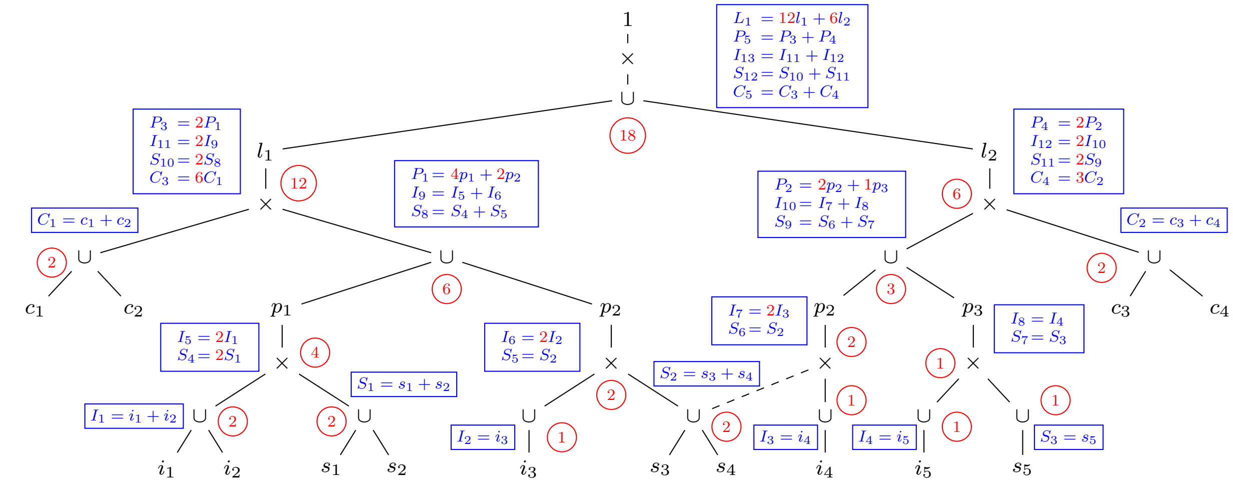
Commutativity with union means that the cofactor matrix for the union of several training datasets is the entry-wise sum of the cofactor matrices of these training datasets. This property is key to the efficiency of our approach, since we can locally compute partial cofactors over different partitions of the training dataset and then add them up. It is also desirable for *concurrent computation*, where partial cofactors can be computed on different cores.

The commutativity with projection implies that we can compute any regression model that consists of a subset of the parameters in the cofactor matrix. During convergence, we simply ignore from the matrix the columns and rows for the irrelevant parameters. This is beneficial if some features are necessary for constructing the dataset but irrelevant for learning, e.g., relation keys supporting the join such as location in our training dataset in Figure 1(a). It is also beneficial for *model selection*, a key challenge in machine learning centered around finding the subset of features that best predict a test dataset. Model selection is a laborious and time-intensive process, since it requires to learn independently parameters corresponding to subsets of the available features. With our reformulation, we first compute the cofactor matrix for all features and then perform convergence on top of the cofactor matrix for the entire lattice of parameters independently of the data. Besides choosing the features after cofactor computation, we may also choose the label and fix its parameter to -1.

The commutativity with projection is crucial for computing the most succinct factorization of the training dataset, since it does not restrict the choice of variable order and allows to retain the join variables in the variable order. Projecting away join variables from the variable order may break the conditional independence of other variables and increase the size of the factorization. Once the factorization is constructed, we may decide to only compute cofactors for a subset of the features.

## 4.3 Factorized Computation of Cofactors

There are several flavors for factorized cofactor computation [19, 14]: over the (non-)materialized factorized dataset or via an optimized SQL query. The materialized flavor is sketched in this section and the SQL flavor is presented in detail in Section 4.4. The underlying idea of all these flavors is that the algebraic factorization used to factorize the training dataset can be mirrored in the factorized computation of the cofactors.



| | $\theta_L$ | $\theta_P$ | $\theta_I$ | $\theta_S$ | $\theta_C$ |
|---|---|---|---|---|---|
| $\Sigma_T$ | $12l_1 + 6l_2$ | $2(4p_1 + 2p_2) + 2(2p_2 + p_3)$ | $4(i_1 + i_2 + i_3 + i_4) + 2i_5$ | $4(s_1 + s_2 + s_3 + s_4) + 2s_5$ | $6(c_1 + c_2) + 3(c_3 + c_4)$ |
| $\Sigma_L$ | $12l_1^2 + 6l_2^2$ | $l_1 P_3 + l_2 P_4$ | $l_1 I_{11} + z_2 I_{12}$ | $l_1 S_{10} + l_2 S_{11}$ | $l_1 C_3 + l_2 C_4$ |
| $\Sigma_P$ | $\Sigma_Z / \theta_P$ | $2(4p_1^2 + 2p_2^2) + 2(2p_2^2 + p_3^2)$ | $2(p_1 I_5 + p_2(I_6 + I_7) + p_3 I_8)$ | $2(p_1 S_4 + p_2(S_5 + S_6) + p_3 S_7)$ | $P_1 C_1 + P_2 C_2$ |
| $\Sigma_I$ | $\Sigma_L / \theta_I$ | $\Sigma_P / \theta_I$ | $4(i_1^2 + i_2^2 + i_3^2 + i_4^2) + 2i_5^2$ | $2(I_1 S_1 + I_2 S_2 + I_3 S_2 + I_4 S_3)$ | $I_9 C_1 + I_{10} C_2$ |
| $\Sigma_S$ | $\Sigma_L / \theta_S$ | $\Sigma_P / \theta_S$ | $\Sigma_I / \theta_S$ | $4(s_1^2 + s_2^2 + s_3^2 + s_4^2) + 2s_5^2$ | $S_8 C_1 + S_9 C_2$ |
| $\Sigma_C$ | $\Sigma_L / \theta_C$ | $\Sigma_P / \theta_C$ | $\Sigma_I / \theta_C$ | $\Sigma_S / \theta_C$ | $6(c_1^2 + c_2^2) + 3(c_3^2 + c_4^2)$ |

**Figure 3: (Top) The factorized join annotated with constant (counts) and linear (weighted sums) aggregates used for cofactor computation. (Bottom) Cofactor matrix based on the annotated factorized join (Column for intercept $T$ not shown, the value for $\Sigma_T/\Theta_T$ is 18). For any model, the convergence of model parameters is run on top of this matrix.**

Example 4.2. Based on our running example in Figure 1(a), consider a linear regression model with all variables as features (the label can be decided later). The cofactor Cofactor$[(P),(I)]$ is the sum of products of features $P$ and $I$. If this aggregate is computed over the factorized join result in Figure 1(d), then we can exploit the algebraic factorization rules

$$\sum_{i=1}^{n} x \to x \cdot n \quad \text{and} \quad \sum_{i=1}^{n} x \cdot a_i \to x \cdot \sum_{i=1}^{n} a_i$$

to obtain

$$\text{Cofactor}[(P),(I)] = 2p_1 \cdot 2(i_1 + i_2) + 2p_2 \cdot 2i_3 + 2p_2 \cdot 2i_4 + 2p_3 \cdot i_5,$$

where the coefficients are the number of $C$-values that are paired with $P$-values and $I$-values. Similarly, we can factorize the pairs of values of independent features as follows:

$$\sum_{i=1}^{r} \sum_{j=1}^{s} (x_i \cdot y_j) \to (\sum_{i=1}^{r} x_i) \cdot (\sum_{j=1}^{s} y_j).$$

For features $P$ and $C$, the cofactor would then be:

$$\text{Cofactor}[(P),(C)] = (4p_1 + 2p_2)(c_1 + c_2) + (2p_2 + p_3)(c_3 + c_4).$$

For a factorization $E$ representing a relation $R$, we compute the cofactors at the root of $E$ using cofactors at children. They require the computation of constant (degree 0) aggregates corresponding to the number of tuples in $R$; linear (degree 1) aggregates for each feature $A$ of $E$, which are sums of all $A$-values, weighted by the number of times they occur in $R$; and quadratic (degree 2) aggregates, which are products of values and/or linear aggregates, or of quadratic and constant aggregates. We call all these aggregates the *regression aggregates* for linear regression models [19].

Figure 3 displays the factorized join result, annotated with the constant (circles) and linear aggregates (rectangles), and an excerpt of the cofactor matrix, whose elements are quadratic aggregates. The $\langle 1 \rangle$ at the top of the factorization represents the intercept of the model. This can be obtained by extending each relation with one attribute $T$ with value 1. The variable orders for the query would then have the variable $T$ as root. □

We generalize the factorized computation of cofactors to polynomial regression models of arbitrary degree $d$. The difference to the linear case is that the cofactors are for more features due to feature interactions and thus the degrees of regression aggregates increase as well. For a given model of degree $d$, the highest-degree aggregates have degree $2d$. We construct high-degree aggregates by combining lower-degree aggregates, following the factorization rules from Example 4.2.

We can compute the cofactors in one pass over a factorized query result.

Proposition 4.3. ([3, 19]) *The cofactors of any polynomial regression model can be computed in one pass over any factorized representation.*

An immediate implication is that the redundancy in the flat join result is not necessary for learning:

Theorem 4.4. *The parameters of any polynomial regression model of degree $d$ can be learned over a query $Q$ and database $\mathbf{D}$ in time $O(n^{2d} \cdot |\mathbf{D}|^{fhtw(Q)} + n^{2d} \cdot s)$, where $n$ is the number of features and $s$ is the number of convergence steps.*

Theorem 4.4 is a direct corollary of Propositions 3.4 and 4.3. Under data complexity, this becomes $O(|\mathbf{D}|^{fhtw(Q)})$ and coincides with the time needed to compute the factorized query result. For computing join queries, this is worst-case optimal within the class of factorized representations over variable orders. The factor $n^{2d}$ is the total number of regression aggregates needed to learn $h_\theta$. In contrast, the data complexity of any regression learner taking a flat join result as input would be at best $O(|\mathbf{D}|^{\rho^*(Q)})$. Furthermore, state-of-the-art learners typically do not decouple parameter convergence from data-dependent computation, so their total runtime to learn $h_\theta$ is $O(n^{2d} \cdot |\mathbf{D}|^{\rho^*(Q)} \cdot s)$.

## 4.4 Factorized Computation in SQL

A SQL encoding of factorized computation of cofactors has two desirable properties. It can be computed by any relational database system and is thus readily deployable in practice with a small implementation overhead. It leverages secondary-storage mechanisms and thus works for databases that do not fit in memory. For lack of space, we focus on learning over a join query $Q_{in}$ (i.e., no projections).

Our approach has two steps. We first rewrite $Q_{in}$ into an ($\alpha$-)acyclic query $Q_{out}$ over possibly cyclic subqueries. Since cyclic queries are not factorizable, we materialize them to new relations. To attain the overall complexity from Theorem 4.4, this materialization requires a worst-case optimal join algorithm like LeapFrog TrieJoin [23]. We then generate a SQL query that encodes the factorized computation of the regression aggregates over $Q_{out}$.

**Rewriting queries with cycles.** Figure 4 gives the rewriting procedure. It works on a variable order $\Delta$ of $Q_{in}$. The set $QS$ is a disjoint partitioning of the set of relations of $Q_{in}$ into sets of relations or partitions. Each partition $Q_A$ is a join query defined by the set $key(A) \cup \{A\}$ of variables. This partition is materialized to a relation with the same name $Q_A$. The materialization simplifies the variable order of $Q_{in}$ to that of an acyclic query $Q_{out}$ equivalent to $Q_{in}$. In case $Q_{in}$ is already acyclic, then each partition has one relation and hence $Q_{out}$ is syntactically equal to $Q_{in}$.

```
rewrite(variable order $\Delta$)
$QS = \emptyset$;    $A = var(\Delta)$;
if ($\Delta = A(\Delta_j)_{j\in[k]}$)   $QS = \bigcup_{j\in[k]}$ rewrite($\Delta_j$);
$Q_A$ = relations($key(A) \cup \{A\}$);
if ($\nexists Q \in QS$ s.t. $Q_A \subseteq Q$)  return  $QS \cup \{Q_A\}$
else   return   $QS$
```

**Figure 4: Rewriting a join query over variable order $\Delta$ into an acyclic join query over possibly cyclic subqueries.**

Example 4.5. Let us consider the bowtie join query and its variable order from Example 3.3. We apply the rewriting algorithm. When we reach leaf $B$ in the left branch, we create the join query $Q_B$ over the relations $\{R_1, R_2, R_3\}$ and add it to $QS$. When we return from recursion to variable $A$, we create the query $Q_A$ over the same relations, so we do not add it to $QS$. We proceed similarly in the right branch: We create the join query $Q_D$ over relations $\{R_4, R_5, R_6\}$ and add it to $QS$. The queries at $E$ and $C$ are not added to $QS$. Whereas the original query and the two subqueries $Q_B$ and $Q_D$ are cyclic, the rewritten query $Q_{out}$ is the join of $Q_B$ and $Q_D$ on $C$ and is acyclic. The triangle queries $Q_B$ and $Q_D$ cannot be computed worst-case optimally with traditional relational query plans [2], but we can use specialized engines to compute them [23].

The join query in Figure 1(a) is already acyclic. Using its variable order from Figure 1(c), we obtain one identity query per relation. □

**SQL query generation**. The algorithm in Figure 5 generates one SQL query that computes all regression aggregates (thus including the cofactors) of a polynomial regression model. It takes as input an extended variable order $\Delta$, which has one extra node per database relation placed under its lowest variable. The query is then constructed in a top-down traversal of $\Delta$.

For a variable order $\Delta$ with root $A$, we materialize a relation $A_{type}$ over the schema $(A_n, A_d)$, where $A_n$ is an identifier for $A$ and $A_d$ encodes the degrees of

```
factorize-sql (extended variable order $\Delta$)

switch($\Delta$) :
  leaf node $R$ :
    CREATE TABLE $R_{type}(R_n, R_d)$;   INSERT INTO $R_{type}$ VALUES $(R, 0)$;
    let
      $deg(G_\Delta) = R_d, \quad lineage(G_\Delta) = (R_n, R_d), \quad agg(G_\Delta) = 1$
    in
      $G_\Delta =$ SELECT $schema(R), lineage(G_\Delta), deg(G_\Delta), agg(G_\Delta)$ FROM $R, R_{type}$;

  inner node $A(\Delta_j)_{j \in [k]}$ :
    CREATE TABLE $A_{type}(A_n, A_d)$;
    foreach $0 \leq i \leq 2d$ do INSERT INTO $A_{type}$ VALUES $(A, i)$;
    foreach $j \in [k]$ do $\mathbf{G}_{\Delta_i} =$ factorize-sql($\Delta_j$);
    let
      $deg(G_\Delta) = \sum_{j \in [k]} deg(G_{\Delta_j}) + A_d, \quad lineage(G_\Delta) = (lineage(G_{\Delta_1}), \ldots, lineage(G_{\Delta_k}), A_n, A_d)$,
      $agg(G_\Delta) = \textbf{sum}(\textbf{power}(A, A_d) * \prod_{j \in [k]} agg(G_{\Delta_j}))$
    in
      $G_\Delta =$ SELECT $key(A), lineage(G_\Delta), deg(G_\Delta), agg(G_\Delta)$
             FROM $G_{\Delta_1}$NATURAL JOIN$\ldots G_{\Delta_k}, A_{type}$
             WHERE $deg(G_\Delta) \leq 2d$ GROUP BY $key(A), lineage(G_\Delta), deg(G_\Delta)$;
return $G_\Delta$;
```

**Figure 5: Generation of one SQL query for computing all regression aggregates used to build a polynomial model over an acyclic query with extended variable order $\Delta$.**

the aggregates over $A$. $A_{type}$ has $2d + 1$ tuples, one for each degree from zero to $2d$.

We generate a query $G_\Delta$, which computes the regression aggregates for the factorization over $\Delta$. $G_\Delta$ is a natural join of the queries $(G_{\Delta_j})_{j \in [k]}$ constructed for the children of $A$ and an inequality join with relation $A_{type}$. The query computes all aggregates ($agg(G_\Delta)$) of degree ($deg(G_\Delta)$) up to $2d$ along with the *lineage* of their computation. These aggregates are computed by combining aggregates computed at children and for $A$. The lineage is given by columns with indices $n$ and $d$. It is a relational encoding of the set $\mathcal{I}$ of indexes of feature interactions as given in Section 4.1. Furthermore, the query retains the variables in $key(A)$ which are to be joined on in queries constructed for the ancestors of $A$.

For a leaf node representing an input relation $R$, the type relation $R_{type}$ has only one tuple $(R, 0)$ and the generated query $G_\Delta$ is a product of $R$ and $R_{type}$. We add a copy of column $R_d$ to represent the degree $deg(G_\Delta)$ and we set $agg(G_\Delta)$ equal to 1. These additions allow us to treat all nodes uniformly.

For simplicity of exposition, we accommodate the intercept $T$ as described in Example 4.2, where each relation is extended with one extra attribute $T$ with value 1. $T$ is used as a root node for any variable order, so variable orders cannot be forests.

Example 4.6. We show how to generate the SQL query for computing the regression aggregates for any polynomial regression model of degree $d$ over the acyclic join in Figure 1(a). The queries constructed at different nodes in the variable order $\Delta$ in Figure 1(c), now extended with three relation nodes, are given in Figure 6.

For the path leading to the Sales node, we generate the queries $Q_{Sales}$, $Q_S$, and $Q_P$ for the nodes Sales, $S$, and respectively $P$.

The query $Q_{Sales}$ constructed at node Sales computes the product of *Sales* and $Sales_{type}$ and adds the degree and aggregate columns.

Variable $S$ only has Sales as child. Its query $Q_S$ computes inequality join on $Q_{Sales}$ and $S_{type}$ to limit the combinations of aggregates to those of degree at most $2d$. It also propagates the lineage from $Q_{Sales}$ and $S_{types}$ and keeps the variable $P$ because it is in its key: $key(S) = \{P\}$.

The query $Q_P$ constructed at variable $P$ computes the natural join of queries $Q_S$ and $Q_I$ and the inequality join with $P_{type}$ on degree. This query also computes the aggregates ($P_{agg}$) with degree ($P_{deg}$) up to $2d$ and their lineage (columns with indexes $n$ and $d$). The query keeps the variable $L$ because it is in its key: $key(P) = \{L\}$. □

**CREATE TABLE** $Q_{Sales}$ **AS**
**SELECT** $\boxed{P, S}$ $\boxed{Sales_n, Sales_d,}$
$\boxed{(Sales_d)}$ **AS** $Sales_{deg}$, $\boxed{1}$ **AS** $Sales_{agg}$
**FROM** $Sales, Sales_{type}$;

**CREATE TABLE** $Q_S$ **AS**
**SELECT** $\boxed{P,}$ $\boxed{Sales_n, Sales_d,}$ $\boxed{S_n, S_d,}$
$\boxed{(Sales_{deg} + S_d)}$ **AS** $S_{deg}$,
$\boxed{\text{sum}(\text{power}(S, S_d) * Sales_{agg})}$ **AS** $S_{agg}$
**FROM** $Q_{Sales}, S_{type}$
**WHERE** $\boxed{(Sales_{deg} + S_d)}$ $<= 2d$
**GROUP BY** $\boxed{P,}$ $\boxed{Sales_n, Sales_d,}$
$\boxed{S_n, S_d, S_{deg}}$

**CREATE TABLE** $Q_P$ **AS**
**SELECT** $\boxed{L,}$ $\boxed{I_n, I_d, Branch_n, Branch_d,}$
$\boxed{S_n, S_d, Sales_n, Sales_d,}$ $\boxed{P_n, P_d,}$
$\boxed{(I_{deg} + S_{deg} + P_d)}$ **AS** $P_{deg}$,
$\boxed{\text{sum}(\text{power}(P, P_d) * I_{agg} * S_{agg})}$ **AS** $P_{agg}$
**FROM** $Q_I$ **NATURAL JOIN** $Q_S, P_{type}$
**WHERE** $\boxed{(I_{deg} + S_{deg} + P_d)}$$<= 2d$
**GROUP BY** $\boxed{L,}$ $\boxed{I_n, I_d, Branch_n, Branch_d,}$
$\boxed{S_n, S_d, Sales_n, Sales_d,}$ $\boxed{P_n, P_d, P_{deg}}$

**Figure 6: Queries generated by the algorithm in Figure 5 at nodes *Sales*, *S*, and *P* in the extended variable order of the rewritten query. In each of these queries, the aggregate, degree, and lineage columns are color-coded.**

## 5. CONCLUSION AND FUTURE WORK

Factorized databases are a fresh look at the problem of computing and representing results to relational queries. So far, we addressed the worst-case optimal computation of factorized results for conjunctive queries under various factorized representation systems for relational data [17], the characterization of the succinctness gap between sizes of factorized and flat representations of query results and their provenance polynomials [16], and the factorized computation of aggregates [3], such as those needed for learning polynomial regression models over database joins [19].

These theoretical results form the foundation of the **FDB** system for query factorization [4, 3], and of the **F** system for learning regression models over factorized queries [19, 14].

We next discuss directions of future research.

**Practical considerations.** We have recently implemented an early prototype for learning regression models and preliminary benchmarks are very encouraging. For linear regression models, our prototype **F** can achieve up to three orders of magnitude performance speed-up over state-of-the-art systems R, Python StatsModels, and MADlib [19]. This performance gap can be further widened by accommodating systems aspects such as compilation of high-level code that only depends on the fixed set of features and not on the arbitrarily large data.

We are currently designing and implementing a second, more robust version of our in-memory query engine **FDB** for factorized databases, whose focus is on a cache-friendly representation and computation and the exploitation of many-cores architectures.

**Beyond polynomial regression.** The principles behind **F** are applicable to learning statistical models beyond least-squares polynomial regression models (with various regularizers) using gradient descent. It works for any model where the derivatives of the objective function are expressible in a semiring with multiplication and summation operations. It also works for classification, such as boosted trees and $k$-nearest neighbors, and other optimization algorithms, such as Newton optimization and coordinate descent. The semirings are necessary, since factorization relies on the commutativity and distributivity laws of semirings.

**Distributed Factorized Computation.** Massively parallel query processing incurs a high network communication cost [21]. This cost has been analyzed theoretically for the Massively-Parallel Communication model, with several recent results on communication optimality for join queries [10].

Since factorized databases were specifically designed for succinct and lossless representations of relational data, a natural idea would be to reduce the communication cost by shuffling factorized, instead of flat, data between computation rounds.

## Acknowledgements

The authors acknowledge Nurzhan Bakibayev, Radu Ciucanu, Tomáš Kočiský, and Jakub Závodný for contributions to various aspects of factorized databases reported in this paper. This work has been supported by an Amazon AWS Research Grant, a Google Faculty Research Award, LogicBlox, and the ERC grant FADAMS agreement 682588.

## 6. REFERENCES

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

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

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

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

[5] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*, 2006.

[6] R. Ciucanu and D. Olteanu. Worst-case optimal join at a time. Technical report, Oxford, Nov 2015.

[7] G. Gottlob. On minimal constraint networks. *Artif. Intell.*, 191-192:42–60, 2012.

[8] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[9] M. A. Khamis, H. Q. Ngo, and A. Rudra. FAQ: Questions Asked Frequently, CoRR:1504.04044, April 2015.

[10] P. Koutris, P. Beame, and D. Suciu. Worst-case optimal algorithms for parallel query processing. In *ICDT*, pages 8:1–8:18, 2016.

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

[12] D. Olteanu and J. Huang. Using OBDDs for efficient query evaluation on probabilistic databases. In *SUM*, pages 326–340, 2008.

[13] D. Olteanu, C. Koch, and L. Antova. World-set decompositions: Expressiveness and efficient algorithms. *TCS*, 403(2-3):265–284, 2008.

[14] D. Olteanu and M. Schleich. F: Regression models over factorized views. *PVLDB*, 9(10), 2016.

[15] D. Olteanu and J. Závodný. On factorisation of provenance polynomials. In *TaPP*, 2011.

[16] D. Olteanu and J. Závodný. Factorised representations of query results: size bounds and readability. In *ICDT*, pages 285–298, 2012.

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

[18] J. Pearl. *Probabilistic reasoning in intelligent systems: Networks of plausible inference.* Morgan Kaufmann, 1989.

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

[20] P. Sen, A. Deshpande, and L. Getoor. Read-once functions and query evaluation in probabilistic databases. *PVLDB*, 3(1):1068–1079, 2010.

[21] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed SQL database that scales. *PVLDB*, 6(11):1068–1079, 2013.

[22] Simons Institute for the Theory of Computing, UC Berkeley. *Workshop on "Succinct Data Representations and Applications"*, September 2013.

[23] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, pages 96–106, 2014.