An Intermediate Representation for Hybrid Database and Machine Learning Workloads

Amir Shaikhha
University of Edinburgh
amir.shaikhha@ed.ac.uk

Maximilian Schleich
University of Washington
schleich@cs.washington.edu

Dan Olteanu
University of Zurich
olteanu@ifi.uzh.ch

ABSTRACT

IFAQ is an intermediate representation and compilation framework for hybrid database and machine learning workloads expressible using iterative programs with functional aggregate queries. We demonstrate IFAQ for several OLAP queries, linear algebra expressions, and learning factorization machines over training datasets defined by feature extraction queries over relational databases.

PVLDB Reference Format:
Amir Shaikhha, Maximilian Schleich, and Dan Olteanu. An Intermediate Representation for Hybrid Database and Machine Learning Workloads. PVLDB, 14(12): 2831 - 2834, 2021.
doi:10.14778/3476311.3476356

1 WHAT IS IFAQ?

The mainstream approach to machine learning over relational data consists of two steps. The training dataset is first constructed via a feature extraction query over the input database using a database system or minimalistic query engines such as Pandas. The desired model is then trained over the result of the query using a statistical software package of choice such as R, scikit-learn, or TensorFlow.

IFAQ is a framework that (1) allows to specify in a unified domain specific language (DSL) both aforementioned tasks and (2) provides a unified optimization and compilation approach to programs in this DSL [9]. Its DSL allows for **I**terative computation of **F**unctional **A**ggregate **Q**ueries to express, e.g., gradient descent optimization for linear regression and factorization machines.

Its optimizations benefit from and enrich three repertoires originally developed in isolation by the database, machine learning, and compiler communities. Each DSL program is subject to several layers of optimizations including: algebraic transformations such as code and data factorization into nested dictionaries [6]; loop transformations; schema specialization; data layout optimizations; and compilation into efficient code specialized for the given workload.

Figure 1 gives a bird's eye view of the IFAQ architecture. IFAQ can interpret the program or generate low-level C++ or Scala code [8]. The latter can be further compiled using the existing backends of Scala to lower-level JS, JVM, or LLVM code. The IFAQ framework can therefore take advantage of existing compilation frameworks that operate at a lower level with optimizations that are orthogonal to IFAQ. The IFAQ code optimizations are applicable to both database and machine learning workloads and tightly interleaves the two. They are layered in a sequence of four transformation blocks as shown in Figure 3. Prior experiments show that they can lead to orders of magnitude performance improvements over mainstream solutions using TensorFlow and scikit-learn [9].

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476356

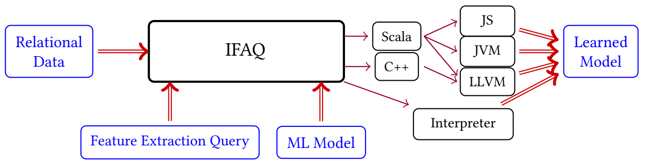


Figure 1: Bird's Eye View of IFAQ Architecture.

We demonstrate IFAQ for database, linear algebra, and hybrid DB/ML workloads. For the latter, we demonstrate the learning process of factorization machines (FMs) [4] for retail forecasting scenarios using commercial and public datasets [1, 5]. FMs model correlations between features as feature interactions, where the parameters for each feature interaction are factorized. They have been successfully applied to retail forecasting [10] and were winning solutions in several Kaggle competitions (e.g., [2]).

2 IFAQ BY EXAMPLE

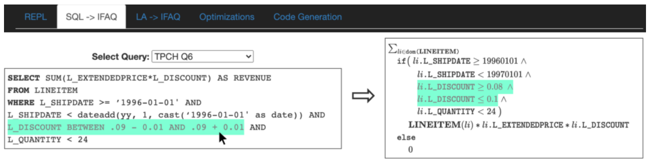
In this section, we present the expressiveness and syntax of IFAQ through examples.

IFAQ for Database Queries. IFAQ represents relations as dictionaries mapping tuples to their multiplicities. We are given a database with three relations: **S**ales (*item, store, units*), Sto**R**es(*store, city*), **I**tems(*item, price*). The join of the three relations $\mathbf{Q} = \mathbf{S} \bowtie \mathbf{R} \bowtie \mathbf{I}$ can be expressed as follows:

```
let Q =
  $\sum_{x_s \in \text{dom}(\mathbf{S})} \sum_{x_r \in \text{dom}(\mathbf{R})} \sum_{x_i \in \text{dom}(\mathbf{I})}$ (
    let k = $\{i = x_s.i, s = x_s.s, c = x_r.c, p = x_i.p\}$ in
    $\{\{k \rightarrow \mathbf{S}(x_s) * \mathbf{R}(x_r) * \mathbf{I}(x_i) * (x_s.i == x_i.i) * (x_s.s == x_r.s)\}\}$
)
```

This expression performs a multi-way join among the three relations, by nested iterations over the elements of each relation (using Σ). The result of the join is expressed as a dictionary, which is constructed using {{key →value}}, where key is the record (constructed using the syntax $\{a = e, ...\}$) and value is the multiplicity. The join condition is expressed by multiplying the join predicate with the multiplicity of each record. Alternatively, one can use conditionals to express predicates as can be seen in Figure 2.

IFAQ for Linear Algebra Workloads. Vectors can be represented as dictionaries from indices to values. Similarly, matrices are dictionaries from indices to vectors. This means that matrices are



```
SELECT SUM(L_EXTENDEDPRICE*L_DISCOUNT) AS REVENUE
FROM LINEITEM
WHERE L_SHIPDATE >= '1996-01-01' AND
L_SHIPDATE < dateadd(yy, 1, cast('1996-01-01' as date)) AND
L_DISCOUNT BETWEEN .09 - 0.01 AND .09 + 0.01 AND
L_QUANTITY < 24
```

```
$\sum_{li \in \mathrm{dom}(\mathbf{LINEITEM})}$
  if( $li.$L_SHIPDATE $\geq 19960101 \wedge$
      $li.$L_SHIPDATE $< 19970101 \wedge$
      $li.$L_DISCOUNT $\geq 0.08 \wedge$
      $li.$L_DISCOUNT $\leq 0.1 \wedge$
      $li.$L_QUANTITY $< 24$ )
    $\mathbf{LINEITEM}(li) * li.$L_EXTENDEDPRICE $* li.$L_DISCOUNT
  else
    0
```

Figure 2: Screenshots of the IFAQ web user interface showing the translation of SQL queries to IFAQ. Users can see the mapping between constructs by hovering over the code snippets.

basically nested dictionaries. As an example, the multiplication of the matrix **A** and vector **V** is as follows:

$$\lambda_{\mathtt{row} \in \mathtt{dom}(\mathbf{A})} \sum_{\mathtt{col} \in \mathtt{dom}(\mathbf{A}(\mathtt{row}))} \mathbf{A}(\mathrm{row})(\mathrm{col}) * \mathbf{V}(\mathrm{col})$$

This expression performs an iteration over the domain of rows of the matrix **A**, and constructs a dictionary with the same row indices. The values of this dictionary are constructed by iterating over the column index col of **A** and then multiplying each element **A**(row)(col) by the colth element of **V**, represented as **V**(col).

IFAQ for In-Database Machine Learning. The goal is to train a machine learning model that predicts u with features $\mathbf{F} = \{i, s, c, p\}$, where the training dataset is given by the join of the three relations $\mathbf{Q} = \mathbf{S} \bowtie \mathbf{R} \bowtie \mathbf{I}$ (cf. Section **??**).

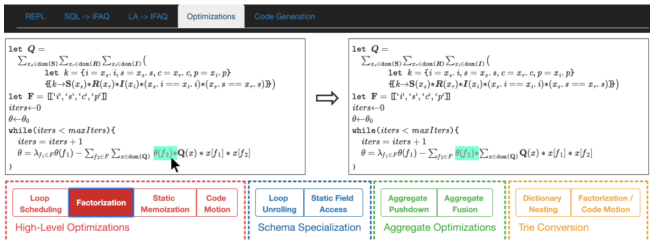
Model. We learn degree-2 factorization machines (FMs) [4] over **Q**. FMs factorize the parameters for feature interactions and implicitly learn a latent vector of rank r for each feature. An FM for a given record x and features **F** is given by the following IFAQ expression:

$$FM(x) = \sum_{f_1 \in \mathbf{F}} \theta(f_1) * x[f_1] + \sum_{f_2 \in \mathbf{F}} \sum_{f_3 \in \mathbf{F}} (f_2 < f_3) * \sum_{k \in \mathbf{L}} \omega(f_2)(k) * \omega(f_3)(k) * x[f_2] * x[f_3]$$

L is the vector of integers from 1 to r, $\boldsymbol{\theta}$ is the vector of the linear parameters, and $\boldsymbol{\omega}$ is the array of the parameters for feature interactions. The condition $(f_2 < f_3)$ ensures that each pairwise feature interaction is considered once.

Batch Gradient Descent (BGD). We learn the model with BGD by repeatedly updating the parameters in the direction of the gradient until convergence. For simplicity of exposition, we make three simplifications: (1) the learning rate is fixed to 1, (2) no regularization, and (3) the convergence criterion is given by a fixed number of BGD iterations. The learning algorithm can be expressed in IFAQ as follows:

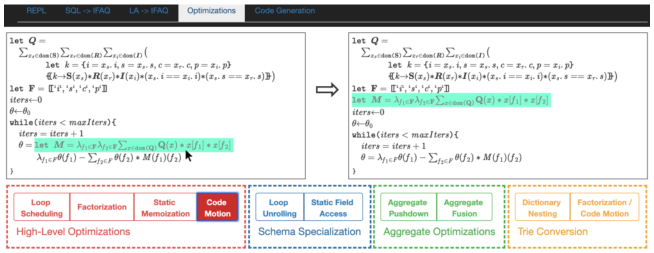
```
let $\mathbf{L} = [\![1..r]\!]$    let $\mathbf{F} = [\![`i`, `s`, `c`, `p`]\!]$
$iters \leftarrow 0$    $\theta \leftarrow \theta_0$    $\omega \leftarrow \omega_0$
while($iters < maxIters$) {
  $iters = iters + 1$
  $\theta = \lambda_{f_1 \in \mathbf{F}} \theta(f_1) - \sum_{x \in \mathtt{dom}(\mathbf{Q})} \mathbf{Q}(x) * x[f_1] * \Big( \sum_{f_2 \in \mathbf{F}} \theta(f_2) * x[f_2] +$
      $\sum_{f_3 \in \mathbf{F}} \sum_{f_4 \in \mathbf{F}} (f_3 < f_4) * \sum_{k \in \mathbf{L}} \omega(f_3)(k) * \omega(f_4)(k) * x[f_3] * x[f_4] \Big)$
  $\omega = \lambda_{f_1 \in \mathbf{F}} \lambda_{\ell \in \mathbf{L}} \omega(f_1)(\ell) - \sum_{f_2 \in \mathbf{F}} (f_1 \neq f_2) *$
      $\sum_{x \in \mathtt{dom}(\mathbf{Q})} \Big( \mathbf{Q}(x) * x[f_1] * x[f_2] * \sum_{f_3 \in \mathbf{F}} \theta(f_3) * x[f_3] +$
      $\sum_{f_3 \in \mathbf{F}} \sum_{f_4 \in \mathbf{F}} (f_3 < f_4) * \sum_{k \in \mathbf{L}} \omega(f_3)(k) * \omega(f_4)(k) * x[f_3] * x[f_4] \Big)$
}
```



```
let $Q =$
  $\sum_{x_s \in \mathtt{dom}(\mathbf{S})} \sum_{x_r \in \mathtt{dom}(R)} \sum_{x_i \in \mathtt{dom}(I)} ($
      let $k = \{i = x_s.i, s = x_s.s, c = x_r.c, p = x_i.p\}$
      $\{\!\{k \rightarrow \mathbf{S}(x_s) * R(x_r) * I(x_i) * (x_s.i == x_i.i) * (x_s.s == x_r.s)\}\!\})$
let $\mathbf{F} = [\![`i`, `s`, `c`, `p`]\!]$
$iters \leftarrow 0$
$\theta \leftarrow \theta_0$
while($iters < maxIters$){
  $iters = iters + 1$
  $\theta = \lambda_{f_1 \in F} \theta(f_1) - \sum_{f_2 \in F} \sum_{x \in \mathtt{dom}(\mathbf{Q})} \theta(f_2) * \mathbf{Q}(x) * x[f_1] * x[f_2]$
}
```

```
let $Q =$
  $\sum_{x_s \in \mathtt{dom}(\mathbf{S})} \sum_{x_r \in \mathtt{dom}(R)} \sum_{x_i \in \mathtt{dom}(I)} ($
      let $k = \{i = x_s.i, s = x_s.s, c = x_r.c, p = x_i.p\}$
      $\{\!\{k \rightarrow \mathbf{S}(x_s) * R(x_r) * I(x_i) * (x_s.i == x_i.i) * (x_s.s == x_r.s)\}\!\})$
let $\mathbf{F} = [\![`i`, `s`, `c`, `p`]\!]$
$iters \leftarrow 0$
$\theta \leftarrow \theta_0$
while($iters < maxIters$){
  $iters = iters + 1$
  $\theta = \lambda_{f_1 \in F} \theta(f_1) - \sum_{f_2 \in F} \theta(f_2) * \sum_{x \in \mathtt{dom}(\mathbf{Q})} \mathbf{Q}(x) * x[f_1] * x[f_2]$
}
```

(a) (left) after applying Loop Scheduling , (right) Factorization moves the expression $\theta(f_2)$ outside the inner sum.



```
let $Q =$
  $\sum_{x_s \in \mathtt{dom}(\mathbf{S})} \sum_{x_r \in \mathtt{dom}(R)} \sum_{x_i \in \mathtt{dom}(I)} ($
      let $k = \{i = x_s.i, s = x_s.s, c = x_r.c, p = x_i.p\}$
      $\{\!\{k \rightarrow \mathbf{S}(x_s) * R(x_r) * I(x_i) * (x_s.i == x_i.i) * (x_s.s == x_r.s)\}\!\})$
let $\mathbf{F} = [\![`i`, `s`, `c`, `p`]\!]$
$iters \leftarrow 0$
$\theta \leftarrow \theta_0$
while($iters < maxIters$){
  $iters = iters + 1$
  $\theta =$ let $M = \lambda_{f_1 \in \mathbf{F}} \lambda_{f_2 \in \mathbf{F}} \sum_{x \in \mathtt{dom}(\mathbf{Q})} \mathbf{Q}(x) * x[f_1] * x[f_2]$
    $\lambda_{f_1 \in F} \theta(f_1) - \sum_{f_2 \in F} \theta(f_2) * M(f_1)(f_2)$
}
```

```
let $Q =$
  $\sum_{x_s \in \mathtt{dom}(\mathbf{S})} \sum_{x_r \in \mathtt{dom}(R)} \sum_{x_i \in \mathtt{dom}(I)} ($
      let $k = \{i = x_s.i, s = x_s.s, c = x_r.c, p = x_i.p\}$
      $\{\!\{k \rightarrow \mathbf{S}(x_s) * R(x_r) * I(x_i) * (x_s.i == x_i.i) * (x_s.s == x_r.s)\}\!\})$
let $\mathbf{F} = [\![`i`, `s`, `c`, `p`]\!]$
let $M = \lambda_{f_1 \in \mathbf{F}} \lambda_{f_2 \in \mathbf{F}} \sum_{x \in \mathtt{dom}(\mathbf{Q})} \mathbf{Q}(x) * x[f_1] * x[f_2]$
$iters \leftarrow 0$
$\theta \leftarrow \theta_0$
while($iters < maxIters$){
  $iters = iters + 1$
  $\theta = \lambda_{f_1 \in F} \theta(f_1) - \sum_{f_2 \in F} \theta(f_2) * M(f_1)(f_2)$
}
```

(b) (left) Static Memoization introduces the variable M, (right) Code Motion moves this variable outside the loop.

Figure 3: Screenshots of the IFAQ web user interface showing the impact of various high-level optimizations on an IFAQ program representing the training of the linear component of the factorization machine (not fully shown to avoid clutter). Lower level optimizations are explained in Section 3.

3 OPTIMIZATIONS

We next show the transformation pipeline for our learning example.

High-Level Optimizations. IFAQ first normalizes the expression by pushing products inside summations. This allows us to reorder the loops in decreasing order of their support, and then factorize the computation to move loop-invariant code out of the inner loops. After these transformations, there is an opportunity to memoize the data-intensive computation over Q and hoist it outside the convergence loop. These computations are represented by $\boldsymbol{M}$, $\boldsymbol{N}$, and $\boldsymbol{O}$ in the following expression:

```
let $\mathbf{L} = [\![1..r]\!]$    let $\mathbf{F} = [\![`i`, `s`, `c`, `p`]\!]$
let $\boldsymbol{M} = \lambda_{f_1 \in \mathbf{F}} \lambda_{f_2 \in \mathbf{F}} \sum_{x \in \mathtt{dom}(\mathbf{Q})} \mathbf{Q}(x) * x[f_1] * x[f_2]$
let $\boldsymbol{N} = \lambda_{f_1 \in \mathbf{F}} \lambda_{f_2 \in \mathbf{F}} \lambda_{f_3 \in \mathbf{F}} \sum_{x \in \mathtt{dom}(\mathbf{Q})} \mathbf{Q}(x) * x[f_1] * x[f_2] * x[f_3]$
let $\boldsymbol{O} = \lambda_{f_1 \in \mathbf{F}} \lambda_{f_2 \in \mathbf{F}} \lambda_{f_3 \in \mathbf{F}} \lambda_{f_4 \in \mathbf{F}} \sum_{x \in \mathtt{dom}(\mathbf{Q})} \mathbf{Q}(x) * x[f_1] * x[f_2] * x[f_3] * x[f_4]$
$iters \leftarrow 0$    $\theta \leftarrow \theta_0$    $\omega \leftarrow \omega_0$
while($iters < maxIters$) {
  $iters = iters + 1$
  $\theta = \lambda_{f_1 \in \mathbf{F}} \theta(f_1) - \sum_{f_2 \in \mathbf{F}} \theta(f_2) * \mathbf{M}(f_1)(f_2) +$
      $\sum_{f_3 \in \mathbf{F}} \sum_{f_4 \in \mathbf{F}} (f_3 < f_4) * \boldsymbol{N}(f_1)(f_3)(f_4) * \sum_{k \in \mathbf{L}} \omega(f_3)(k) * \omega(f_4)(k)$
  $\omega = \lambda_{f_1 \in \mathbf{F}} \lambda_{\ell \in \mathbf{L}} \omega(f_1)(\ell) - \sum_{f_2 \in \mathbf{F}} (f_1 \neq f_2) * \sum_{f_3 \in \mathbf{F}} \theta(f_3) * \boldsymbol{N}(f_1)(f_2)(f_3) +$
      $\sum_{f_3 \in \mathbf{F}} \sum_{f_4 \in \mathbf{F}} (f_3 < f_4) * \boldsymbol{O}(f_1)(f_2)(f_3)(f_4) * \sum_{k \in \mathbf{L}} \omega(f_3)(k) * \omega(f_4)(k)$
}
```

A more detailed impact of each individual optimization of this phase is shown in Figure 3.

Schema Specialization. Dictionaries with statically-known keys of field type can be converted into records in two stages.

During Loop Unrolling , IFAQ unrolls loops over statically-known ranges. For example, ***M*** is unrolled as follows:

$$\texttt{let}\ \mathbf{M} = \{\{`i` \to \{\{\dots, `c` \to \sum_{x\in\texttt{dom}(\mathbf{Q})} \mathbf{Q}(x)*x[`i`]*x[`c`], \dots\}\}, \dots\}\}$$

For Static Field Access , the unrolled dictionaries with keys of field type are turned into records and all dynamic field accesses are turned into static ones. For instance, ***M*** is turned into the nested record *M*:

$$\texttt{let}\ M = \{i = \{\dots, c = \sum_{x\in\texttt{dom}(\mathbf{Q})} \mathbf{Q}(x)*x.i*x.c, \dots\}, \dots\}$$

In the following, we focus on the data-intensive computation of *M*, the records *N* and *O* are treated similarly.

Aggregate Optimization. Each entry in *M* requires the computation of one aggregate over the join result (**Q**). To speedup the computation of *M*, IFAQ interleaves the computation of the entire aggregate batch and the join computation, which is reminiscent of aggregate pushdown in database systems. To demonstrate how IFAQ achieves this, we focus on the computation of one entry $M_{c,p}$ which is defined as:

$$\begin{aligned}
&\texttt{let}\ M_{c,p} = \sum_{x\in\texttt{dom}(\mathbf{Q})} \mathbf{Q}(\text{x}) * \text{x.c} * \text{x.p}\\
&\texttt{let}\ \text{M} = \{\text{c}=\{\dots, \text{p}=M_{c,p},\dots\}\ ,\dots\}\ \dots
\end{aligned}$$

In Aggregate Pushdown , IFAQ first fuses the construction of **Q** with its consumption as follows:

$$\begin{aligned}
&\texttt{let}\ M_{c,p} =\\
&\sum_{x_s\in\texttt{dom}(\mathbf{S})}\ \sum_{x_r\in\texttt{dom}(\mathbf{R})}\ \sum_{x_i\in\texttt{dom}(\mathbf{I})} (\\
&\quad \texttt{let}\ \text{k} = \{i = x_s.i, s = x_s.s, c = x_r.c, p = x_i.p\}\ \texttt{in}\\
&\quad \texttt{let}\ \mathbf{Q} = \{\{\text{k} \to \mathbf{S}(x_s)*\mathbf{R}(x_r)*\mathbf{I}(x_i)*\\
&\qquad\qquad (x_s.\text{i}{=}{=}x_i.\text{i})*(x_s.\text{s}{=}{=}x_r.\text{s})\}\}\ \texttt{in}\\
&\sum_{x\in\texttt{dom}(\mathbf{Q})} \mathbf{Q}(\text{x}) * \text{x.c} * \text{x.p}\\
&)
\end{aligned}$$

As the inner sum iterates over a singleton dictionary, it can be optimized as follows:

$$\begin{aligned}
&\texttt{let}\ M_{c,p} =\\
&\quad\sum_{x_s\in\texttt{dom}(\mathbf{S})}\ \sum_{x_r\in\texttt{dom}(\mathbf{R})}\ \sum_{x_i\in\texttt{dom}(\mathbf{I})} (\\
&\quad \texttt{let}\ \text{k} = \{i = x_s.i, s = x_s.s, c = x_r.c, p = x_i.p\}\ \texttt{in}\\
&\quad \texttt{let}\ \mathbf{Q} = \{\{\text{k} \to \mathbf{S}(x_s)*\mathbf{R}(x_r)*\mathbf{I}(x_i)*\\
&\qquad\qquad (x_s.\text{i}{=}{=}x_i.\text{i})*(x_s.\text{s}{=}{=}x_r.\text{s})\}\}\ \texttt{in}\\
&\quad \texttt{let}\ \text{x} = \text{k}\ \texttt{in}\ \mathbf{Q}(\text{x}) * \text{x.c} * \text{x.p}\\
&)
\end{aligned}$$

Then, IFAQ applies partial evaluation transformations such as inlining field accesses and retrieving the value of a singleton dictionary, which results in the following expression:

$$\begin{aligned}
&\texttt{let}\ M_{c,p} =\\
&\quad\sum_{x_s\in\texttt{dom}(\mathbf{S})}\ \sum_{x_r\in\texttt{dom}(\mathbf{R})}\ \sum_{x_i\in\texttt{dom}(\mathbf{I})} (\\
&\qquad \mathbf{S}(x_s)*\mathbf{R}(x_r)*\mathbf{I}(x_i)*(x_s.\text{i}{=}{=}x_i.\text{i})*(x_s.\text{s}{=}{=}x_r.\text{s}) * x_r.\text{c} * x_i.\text{p}\\
&)
\end{aligned}$$

As the next step, IFAQ leverages the distributivity of multiplication over addition and factorizes the operands of multiplication as follows:

$$\begin{aligned}
&\texttt{let}\ M_{c,p} =\\
&\quad\sum_{x_s\in\texttt{dom}(\mathbf{S})} \mathbf{S}(x_s)*\\
&\qquad\sum_{x_r\in\texttt{dom}(\mathbf{R})} \mathbf{R}(x_r)*(x_s.\text{s}{=}{=}x_r.\text{s}) * x_r.\text{c} *\\
&\qquad\quad\sum_{x_i\in\texttt{dom}(\mathbf{I})} \mathbf{I}(x_i)*(x_s.\text{i}{=}{=}x_i.\text{i}) * x_i.\text{p}
\end{aligned}$$

By performing static memoization and loop-invariant code motion, IFAQ creates the views $\boldsymbol{V_R}$ and $\boldsymbol{V_I}$ that are responsible for computing partial aggregates before join:

$$\begin{aligned}
&\texttt{let}\ V_R = \sum_{x_r\in\texttt{dom}(\mathbf{R})} \mathbf{R}(x_r) * \{\{\{\text{s}{=}x_r.\text{s}\} \to x_r.\text{c}\}\}\\
&\texttt{let}\ V_I = \sum_{x_i\in\texttt{dom}(\mathbf{I})} \mathbf{I}(x_i) * \{\{\{\text{i}{=}x_i.\text{i}\} \to x_i.\text{p}\}\}\\
&\texttt{let}\ M_{c,p} = \sum_{x_s\in\texttt{dom}(\mathbf{S})} \mathbf{S}(x_s) * V_R(\{\text{s}{=}x_s.\text{s}\}) * V_I(\{\text{i}{=}x_s.\text{i}\})
\end{aligned}$$

We perform a similar process for:

$$\texttt{let}\ M_{c,c} = \sum_{x\in\texttt{dom}(\mathbf{Q})} \mathbf{Q}(\text{x}) * \text{x.c} * \text{x.c}$$

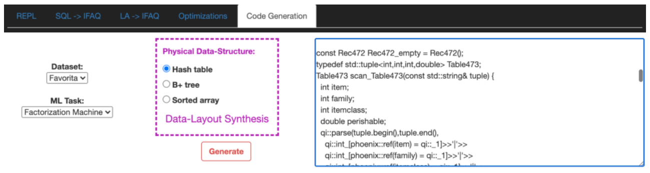
IFAQ similarly pushes the aggregates as follows:

$$\begin{aligned}
&\texttt{let}\ V'_R = \sum_{x_r\in\texttt{dom}(\mathbf{R})} \mathbf{R}(x_r) * \{\{\{\text{s}{=}x_r.\text{s}\} \to x_r.\text{c} * x_r.\text{c}\}\}\\
&\texttt{let}\ V'_I = \sum_{x_i\in\texttt{dom}(\mathbf{I})} \mathbf{I}(x_i) * \{\{\{\text{i}{=}x_i.\text{i}\} \to 1\}\}\\
&\texttt{let}\ M_{c,c} = \sum_{x_s\in\texttt{dom}(\mathbf{S})} \mathbf{S}(x_s) * V'_R(\{\text{s}{=}x_s.\text{s}\}) * V'_I(\{\text{i}{=}x_s.\text{i}\})
\end{aligned}$$

Aggregate Fusion merges the expressions for these two aggregates, and results in the following expression:

$$\begin{aligned}
&\texttt{let}\ \boldsymbol{W_R} = \sum_{x_r\in\texttt{dom}(\mathbf{R})} \mathbf{R}(x_r) *\\
&\qquad \{\{\{\text{s}{=}x_r.\text{s}\} \to \{v_R{=}x_r.\text{c}, v'_R{=}x_r.\text{c} * x_r.\text{c}\}\}\}\\
&\texttt{let}\ \boldsymbol{W_I} = \sum_{x_i\in\texttt{dom}(\mathbf{I})} \mathbf{I}(x_i) *\\
&\qquad \{\{\{\text{i}{=}x_i.\text{i}\} \to \{v_I{=}x_i.\text{p}, v'_I{=}1\}\}\}\\
&\texttt{let}\ M_{cc,pc} = \sum_{x_s\in\texttt{dom}(\mathbf{S})} \mathbf{S}(x_s) * (\\
&\qquad \texttt{let}\ w_R = \boldsymbol{W_R}(\{\text{s}{=}x_s.\text{s}\})\\
&\qquad \texttt{let}\ w_I = \boldsymbol{W_I}(\{\text{i}{=}x_s.\text{i}\})\\
&\qquad \{m_{c,p} = w_R.v_R*w_I.v_I,\ m_{c,c} = w_R.v'_R*w_I.v'_I\}\ )\\
&\texttt{let}\ M_{c,p} = M_{cc,pc}.m_{c,p}\ \texttt{let}\ M_{c,c} = M_{cc,pc}.m_{c,c}
\end{aligned}$$

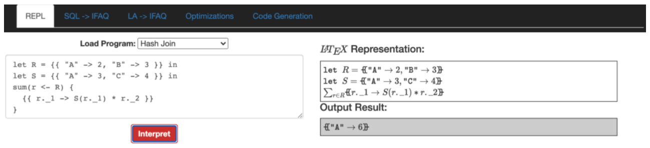
Trie Conversion. Instead of representing each relation and intermediate view as a listing representation, IFAQ represents them as nested collections, which represent tries that are nested by join attributes. Let us focus on the computation of $M_{cc,pc}$.



Dataset: Favorita
ML Task: Factorization Machine
Physical Data-Structure: [✓ Hash table] B+ tree Sorted array
Data-Layout Synthesis

```
const Rec472 Rec472_empty = Rec472();
typedef std::tuple<int,int,int,double> Table473;
Table473 scan_Table473(const std::string& tuple) {
 int item;
 int family;
 int itemclass;
 double perishable;
 qi::parse(tuple.begin(),tuple.end(),
  qi::int_[phoenix::ref(item) = qi::_1]>>'|'>>
  qi::int_[phoenix::ref(family) = qi::_1]>>'|'>>
```

Figure 4: Screenshots of the IFAQ web user interface for code generation. Users can choose different data-layout options, and inspect the generated C++ code.



Load Program: Hash Join

```
let R = {{ "A" -> 2, "B" -> 3 }} in
let S = {{ "A" -> 3, "C" -> 4 }} in
sum(r <- R) {
  {{ r._1 -> S(r._1) * r._2 }}
}
```

LaTeX Representation:

let $R = \{\{"A" \to 2, "B" \to 3\}\}$
let $S = \{\{"A" \to 3, "C" \to 4\}\}$
$\sum_{r \in R} \{\{r.\_1 \to S(r.\_1) * r.\_2\}\}$

Output Result:

$\{\{"A" \to 6\}\}$

Figure 5: IFAQ programs can be authored ad hoc and are evaluated in an interactive read-eval-print loop (REPL) in the web browser.

Dictionary Nesting represents S as a nested dictionary S' instead of iterating over the domain of the keys of S. The nested dictionary S' contains the domain of the field s at its first level, and the domain of field i at its second level. We then iterate over it following the hierarchy in the key:

```
let $M_{cc,pc}$ =
  $\sum_{x_s \in \mathrm{dom}(S')}$
    $\sum_{x_i \in \mathrm{dom}(S'(x_s))}$  $S'(x_s)(x_i) * ($
          let $w_R = \boldsymbol{W}_R(\{s=x_s.s\})$
          let $w_I = \boldsymbol{W}_I(\{i=x_i.i\})$
          $\{m_{c,p} = w_R.v_R * w_I.v_I,$
            $m_{c,c} = w_R.v'_R * w_I.v'_I\}\ )$
```

Factorization / Code Motion This transformation enables more opportunities for factorizing and then hoisting the computation outside the introduced nested summations. For instance, we can hoist the let binding for $\mathbf{W}_R$ and the computation over w_R out of the loop over x_i:

```
let $M_{cc,pc}$ =
  $\sum_{x_s \in \mathrm{dom}(S')}$
    let $w_R = \boldsymbol{W}_R(\{s=x_s.s\})$
    $\{m_{c,p} = w_R.v_R, m_{c,c} = w_R.v'_R\} *$
      $\sum_{x_i \in \mathrm{dom}(S'(x_s))}$  $S'(x_s)(x_i) * ($
          let $w_I = \boldsymbol{W}_I(\{i=x_i.i\})$
          $\{m_{c,p} = w_I.v_I, m_{c,c} = w_I.v'_I\}\ )$
```

Data-Layout Synthesis. As the final step, IFAQ chooses the data layout and generates low-level C++ or Scala code similarly to existing query compilers [7, 8]. IFAQ makes low-level design decisions, including choosing the physical data structure for each dictionary.

Physical Data-Structure IFAQ currently supports hash tables, tree-based, and sorted dictionaries. Each of these data structures show advantages for different workloads.

4 DEMONSTRATION SCENARIOS

Users can interact with the IFAQ web interface.

REPL. To better familiarize users with the syntax and semantics of IFAQ, we provide an interactive read-eval-print loop (REPL). Several sample IFAQ programs (DB queries and LA expressions) are provided as starting point. The user can modify these programs in the textbox, and the LaTeX representation of that program is shown (using the MathJax library). Furthermore, as the IFAQ framework is implemented in Scala, the program is parsed, type-checked, and interpreted in the web-browser thanks to Scala.JS. Finally, the output result of interpreting the program is shown to the user (Figure 5).

SQL and LA to IFAQ. We showcase the expressive power of IFAQ for database and linear algebra workloads by demonstrating the translation of a handful of SQL queries (Figure 2) and simple linear algebra expressions such as vector and matrix operations.

Optimizations. The multi-layer optimizations of IFAQ are demonstrated using a web interface (Figure 3). These optimizations are presented on IFAQ expressions for the feature extraction query and the gradient descent optimization for training factorization machines. To avoid the clutter, we demonstrate the linear components of this model only, which is the same as linear regression. The web interface shows the IFAQ expression before and after applying each optimization. As optimizations require additional dataset information (e.g., variable ordering [9]), we only show these optimizations for the dataset used in Section 2.

The web interface uses code provenance across transformations. When a user hovers over a snippet of code before the optimization, the corresponding code snippet after applying the optimization is highlighted. Similarly, users can interact with the web interface for SQL and LA translations.

Code Generation. The users can select different data-layout synthesis options, and inspect the generated C++ code (Figure 4). We use the predefined IFAQ programs of training factorization machines and linear regression models over relational data. We consider two real-world datasets used in retail forecasting scenarios: 1) *Favorita* [1], a publicly available Kaggle dataset, and 2) *Retailer*, a dataset from a commercial retailer [5].

REFERENCES

[1] C. Favorita. Corp. Favorita Grocery Sales Forecasting: Can you accurately predict sales for a large grocery chain?, October 2017.
[2] Y. Juan, Y. Zhuang, W.-S. Chin, and C.-J. Lin. Field-aware factorization machines for ctr prediction. In *RecSys*, pages 43–50, 2016.
[3] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.
[4] S. Rendle. Factorization machines. In *ICDM*, pages 995–1000. IEEE, 2010.
[5] M. Schleich, D. Olteanu, M. Abo Khamis, H. Ngo, and X. Nguyen. A layered aggregate engine for analytics workloads. In *SIGMOD*, pages 1642–1659, 2019.
[6] A. Shaikhha, M. Huot, J. Smith, and D. Olteanu. Functional collection programming with semi-ring dictionaries. *arXiv preprint arXiv:2103.06376*, 2021.
[7] A. Shaikhha, Y. Klonatos, and C. Koch. Building efficient query engines in a high-level language. *TODS*, 43(1):1–45, 2018.
[8] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch. How to architect a query compiler. In *SIGMOD*, pages 1907–1922, 2016.
[9] A. Shaikhha, M. Schleich, A. Ghita, and D. Olteanu. Multi-layer optimizations for end-to-end data analytics. In *CGO*, page 145–157, 2020.
[10] M. Yurochkin, X. Nguyen, and N. Vasiloglou. Multi-way interacting regression via factorization machines. In *NIPS*, pages 2598–2606, 2017.