ness to perform aggressive optimizations (such as rewriting linear algebra expressions). However, more recent systems, such as *RIOT* [381, 385], *pandas*, and *Spark*, while remaining interpreted, use deferred evaluation to build up larger expressions and optimize them much like relational queries. Chapter 4 will provide a more in-depth discussion of such optimization scopes.

Roadmap: We begin with a series of examples of implementing ML algorithms in database systems. The approaches range from simply leveraging SQL, to progressively more complex uses of the standard database extension mechanisms of *UDTs*, *UDFs*, and *UDAs* (*User-Defined Types, Functions, and Aggregates*), to modifying the language and internals of a database system to support ML. While some of these approaches discussed in this chapter have started showing their age, they still serve as good background for understanding the opportunities and limitations for doing ML within database systems. We conclude this chapter with a broader discussion of how various ML systems leverage database systems or techniques; some of these approaches will be covered in more detail by subsequent chapters. We also point out connections to several topics that the database research community has worked on.

2.1 LINEAR ALGEBRA

Many ML algorithms can be naturally expressed in the language of matrices and linear algebra. In this section, we explore ways to represent matrices as database tables, and to express linear algebra operations in SQL. To illustrate, let us consider a simple but interesting problem—supporting matrix-matrix multiply in SQL. We shall start with the most straightforward approach, and then examine a series of improvements.

Example 2.1 Matrix Multiply with Sparse Representation Consider the product $\mathbf{C}$ of an $m \times \ell$ matrix $\mathbf{A}$ and an $\ell \times n$ matrix $\mathbf{B}$, where $c_{ij} = \sum_{k=1}^{\ell} a_{ik} b_{kj}$. Suppose we represent each matrix using a separable table whose tuples store individual elements of the matrix, i.e., $\mathbf{A}$ is represented by a table `A(i, j, val)`, where `val` stores the value of the element positioned at `(i, j)`.[1] With this representation, we can compute $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ using the following SQL query:

```
SELECT A.i, B.j, SUM(A.val*B.val)
FROM A, B
WHERE A.j = B.i
GROUP BY A.i, B.j;
```

Note that for a sparse matrix, we can omit its 0-valued elements in its table representation, and the above SQL query still works. Also, by specifying the order of attributes `i` and `j` when declaring the primary key for the table, we can simulate either row-major `(i, j)` or column-major storage `(j, i)` order for the matrix.

[1] On a related note, while this triple-based sparse representation is quite natural from the perspective of a matrix, data in matrices often start out in database tables where individual features are stored as columns. In that case, some preprocessing (using an "unpivot" operation) would be needed to convert data into this sparse representation.

Another technicality worth noting is that when writing element-wise operations (e.g., $\mathbf{A} + \mathbf{B}$) in SQL under this representation, a full outer join would be required. Had we used an inner join instead in that case, we would lose result elements at positions where one of the input matrix has a 0 element. ■

This straightforward approach, considered in *RIOT-DB* [381] and *MAD* [81], works quite well for sparse matrices, but it is rather inefficient for dense matrices. First, we need to store two extra integers `i` and `j` for each element, resulting in considerable storage (and hence processing) overhead compared with a more compact, array-based representation that requires no explicit storage of `i` and `j`. Second, when executing the above query, a database system will likely be unable to match the performance of highly optimized linear algebra libraries such as BLAS. On the other hand, despite these inefficiencies, this straightforward approach can still beat implementations that only optimize in-memory computation when data cannot all fit in main memory, thanks to database systems' built-in support for massive data.

We can further eliminate the inefficiencies noted above using the extensibility features found in modern database systems, which allow users to define new, more complex data types and functions (UDTs and UDFs, respectively). The UDFs can call highly optimized linear algebra libraries for computation. For example, by defining a UDT for vectors and a UDF that efficiently computes the dot product of two vectors, we arrive at the following approach.

Example 2.2 Matrix Multiply with Vector Representation Using a vector UDT, we represent matrix $\mathbf{A}$ as a table `A(i, row)`, where the vector-typed `row` stores the `i`-th row of $\mathbf{A}$. Similarly, we represent matrix $\mathbf{B}$ as a table `B(j, col)`, where the vector-typed `col` stores the `j`-th column of $\mathbf{B}$. Suppose the UDF `dotprod`$(\mathbf{v}_1, \mathbf{v}_2)$ computes the dot product of two vectors. Then, we can compute $\mathbf{A} \times \mathbf{B}$ using the following SQL query:

```
SELECT A.i, B.j, dotprod(A.row, B.col)
FROM A, B;
```

Note that we can encapsulate a considerable amount of optimization inside the UDT and UDF. For example, depending on the sparsity of a vector, it can be stored using either densely (as an array of values) or sparsely (as pairs of indices and non-zero values); `dotprod` can operate differently based on the input representations, and calling the appropriate BLAS routines for computation. ■

This approach, considered also in *MAD* [81] and used in many systems such as *Bismarck* [115] and *BUDS* [118], avoids the two inefficiencies of the approach in Example 2.1, by using a more compact representation for dense matrices and by leveraging highly optimized libraries. However, this approach is not without its own issues. First, it exposes different representations to users—$\mathbf{A}$ is by row, $\mathbf{B}$ is by column, while $\mathbf{A} \times \mathbf{B}$ is produced in a sparse representation. It is not difficult to write SQL code to convert between representations, but having to do so runs counter to the mantra of "physical data independence" endeared by database systems.

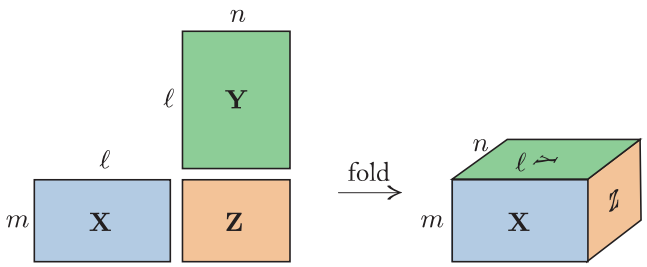

Figure 2.1: Illustration of the compute-to-I/O ratio for matrix multiply $\mathbf{Z} = \mathbf{X} \times \mathbf{Y}$. The volume of the cuboid on the right is proportional to the amount of computation while the surface area is proportional to the amount of I/O.

The second issue is performance. Multiplying rows and columns may not be the most efficient way to multiply two matrices. To see why, note that multiplying an $m \times \ell$ matrix with an $\ell \times n$ matrix involves $\ell \times m \times n$ element-wise multiplications (assuming a simple algorithm, not those with subcubic running time such as Strassen), as well as reading/writing $m \times \ell + \ell \times n + n \times m$ elements. As illustrated in Figure 2.1, these two quantities correspond to the volume and (one half of) the surface area of a cuboid formed by "folding" the two input matrices together with its product. When multiplying two large matrices that do not fit in available fast memory, we need to partition them into submatrices to multiply—which can be viewed as dividing the big cuboid into small cuboids to process. Processing each small cuboid requires I/Os proportional to its surface area. Intuitively, to minimize I/Os, we would like to minimize the surface-to-volume ratio of these cuboids, by making them as close to a perfect cube as possible; a $c \times c \times c$ cube has a surface-to-volume ratio of only $\Theta(c^2/c^3) = \Theta(1/c)$. In contrast, multiplying rows by columns would result in cuboids of dimension $1 \times \ell \times 1$, with a surface-to-volume ratio as high as $\Theta(1)$—in other words, we are not getting much computation done for each chunk of data we bring into fast memory.

To resolve the above issues, we need a more flexible matrix representation that can give us "blockier" submatrices, as used in *RIOT* [381] and *SimSQL* [221].

Example 2.3 Matrix Multiply with Blocked Representation Suppose we have a UDT for (small) matrices or 2d arrays, a UDF `matmult` that multiplies two matrices of compatible dimensions represented in this UDT, and a UDA `matsum` that (element-wise) sums up matrices of identical dimensions represented in this UDT. To represent a large matrix, we divide it into a collection of blocks (submatrices), and store them in a table (`i`, `j`, `block`), where the UDT-typed `block` stores the submatrix in the `i`-th row and `j`-th column (of the submatrices). Given **A** and **B** represented this way using blocks of compatible dimensions, we can compute $\mathbf{A} \times \mathbf{B}$ using the following SQL query:

```
SELECT A.i, B.j, matsum(matmult(A.block, B.block))
FROM A, B
WHERE A.j = B.i
GROUP BY A.i, B.j;
```

Typically, we make the blocks square-shaped, identically sized, and large enough for efficient data transfer and to let UDFs such as `matmult` take advantage of high-performance non-SQL library routines. Some padding and/or additional metadata may be needed to handle matrices whose dimensions are not perfect multiples of the default. While non-squared, non-uniform blocking is theoretically possible, in practice it becomes very complicated because it needs to handle possibly different block dimensions alignment across matrices. ■

A traditional database system will do pretty well with the above approach, but there is still room for improvement. For example, given enough fast memory to accommodate many blocks, we should group multiple blocks into larger square submatrices to multiply in memory. A traditional database optimizer will fail to recognize this grouping strategy because it has no knowledge of the semantics of matrix multiply. Teaching a database (or database-style) optimizer how to handle linear algebra will be a topic we cover in Chapter 4. We will also revisit matrix blocking in Chapter 5 when discussing data-parallel execution, and Chapter 6 when discussing access methods.

While we have considered only matrix multiply thus far, many other linear algebra operations can be handled by SQL with similar approaches. SQL queries can be composed together to process complex linear algebra expressions.

Example 2.4 Ordinary Least Squares We are given data $(\mathbf{X}, \mathbf{y})$ consisting of n observations, where the i-th observation includes a response y_i and values of p predictors $x_{i1}, x_{i2}, \ldots, x_{ip}$. Consider a linear regression model $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$, where $\boldsymbol{\beta}$ is a $p \times 1$ vector of parameters that we wish to learn from data, and $\boldsymbol{\epsilon}$ is an $n \times 1$ vector of errors. Suppose we wish to find $\boldsymbol{\beta}$ to minimize the sum of squared errors, i.e., $(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^\top(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})$. Under a number of reasonable assumptions, the optimal $\boldsymbol{\beta}$ value, called the *Ordinary Least Squares* estimator, can be computed explicitly by:

$$(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}.$$

This linear algebra expression involves only multiplies, transposes, and an inverse. We have already seen how to do matrix multiplies in SQL; matrix-vector multiply is similar. Transpose is straightforward in SQL. Here, inverse is applied to $\mathbf{X}^\top\mathbf{X}$, a $p \times p$ matrix, which is likely not too big as p denotes the number of predictors/parameters. Thus, we can treat $\mathbf{X}^\top\mathbf{X}$ as a "value" and use a UDF `inverse` that invokes a non-SQL library routine to invert a memory-resident matrix [81, 141, 221]. ■

As the above example illustrates, many linear algebra operations are conveniently expressible in SQL with a little help from "scalar" UDFs—i.e., those that operate on individual values

(possibly of UDTs). However, there are also other operations that are harder to express. For example, inverting a large matrix using SQL is more difficult. To invert a matrix $\mathbf{M}$ in a block-wise fashion, let $\mathbf{M} = \left[\begin{smallmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{smallmatrix}\right]$. Then, inverting $\mathbf{M}$ can be reduced to inverting $\mathbf{A}$ (which can be done in memory by choosing a small enough $\mathbf{A}$) and then inverting $(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})$, the Schur complement of $\mathbf{A}$ (which has the same size as $\mathbf{D}$ and can be solved recursively). Coding this procedure would require some recursion or looping construct, which still can be done using more powerful, non-scalar UDFs that can operate on whole database tables, but some of SQL's simplicity and optimizability would be lost.

2.2 ITERATIVE ALGORITHMS

As discussed at the end of the last section, we always have the option of implementing complex algorithms using UDFs coded in procedural extensions of SQL or even other languages such as Python or R. Oftentimes, it is beneficial for UDFs to limit themselves to providing high-level execution control, while pushing much of data processing down into the database system by issuing appropriate queries; such UDFs are called "driver" UDFs in [141]. Many iterative algorithms in ML can be implemented as driver UDFs.

Beyond driver UDFs, certain computation patterns commonly found in ML, such as iterating over a large dataset, can leverage better support from database systems. In particular, UDAs have shown to be a surprisingly flexible tool when implementing ML algorithms inside database systems. Briefly, a user can define a UDA by specifying three functions.

- `Init`(*state*) initializes the state so it is ready to receive data to aggregate. For example, to implement `AVG` as a UDA, we will maintain the sum s and count c of data items seen thus far, and initialize the state (s, c) as $(0, 0)$.

- `Accumulate`(*state*, *data*) updates the state with the new *data* item. For example, for `AVG`, we simply increment s by the value of *data* and increment c by 1 (ignoring `NULL` values).

- `Finalize`(*state*) computes the final result from the state. For example, for `AVG`, we return s/c (or `NULL` if $c = 0$).

Optionally, if the aggregation can be computed over different subsets of the data independently and combined together, the user can supply a fourth function.

- `Merge`($state_1, state_2$) merges state values computed over disjoint input subsets into one. For example, for `AVG`, we merge (s_1, c_1) and (s_2, c_2) into $(s_1 + s_2, c_1 + c_2)$.

`Merge` enables a database system to perform additional optimizations automatically, e.g., partitioning the input data and executing the UDA in parallel.