BlockJoin: Efficient Matrix Partitioning Through Joins

Andreas Kunft[*] Asterios Katsifodimos[†] Sebastian Schelter[*]
Tilmann Rabl[*‡] Volker Markl[*‡]

[*]Technische Universität Berlin
first.lastname@tu-berlin.de

[‡]German Research Center for Artificial Intelligence (DFKI)
first.lastname@dfki.de

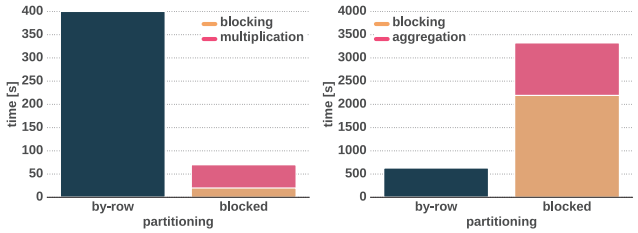
[†]SAP Innovation Center
first.lastname@sap.com

ABSTRACT

Linear algebra operations are at the core of many Machine Learning (ML) programs. At the same time, a considerable amount of the effort for solving data analytics problems is spent in data preparation. As a result, end-to-end ML pipelines often consist of (*i*) relational operators used for joining the input data, (*ii*) user defined functions used for feature extraction and vectorization, and (*iii*) linear algebra operators used for model training and cross-validation. Often, these pipelines need to scale out to large datasets. In this case, these pipelines are usually implemented on top of dataflow engines like Hadoop, Spark, or Flink. These dataflow engines implement relational operators on *row-partitioned* datasets. However, efficient linear algebra operators use *block-partitioned* matrices. As a result, pipelines combining both kinds of operators require rather expensive changes to the physical representation, in particular re-partitioning steps. In this paper, we investigate the potential of reducing shuffling costs by fusing relational and linear algebra operations into specialized physical operators. We present *BlockJoin*, a distributed join algorithm which directly produces block-partitioned results. To minimize shuffling costs, BlockJoin applies database techniques known from columnar processing, such as index-joins and late materialization, in the context of parallel dataflow engines. Our experimental evaluation shows speedups up to 6× and the skew resistance of BlockJoin compared to state-of-the-art pipelines implemented in Spark.

1. INTRODUCTION

Requirements for data analytics applications based on machine learning techniques have changed over the last years. End-to-end ML pipelines nowadays go beyond pure linear algebra and often also include data preparation and transformation steps (ETL) that are best defined using relational algebra operators. Data scientists construct feature-vector representations for training ML models by filtering, joining, and transforming datasets from diverse data sources [39] on a daily basis. This process is often repeated many times in an *ad-hoc* fashion, as a variety of features are explored and selected for optimal predictive performance.



(a) Matrix multiplication. (b) Mean of each row.

Figure 1: Performance implications of row- vs. block-partitioning in two linear algebra operators.

Such pipelines are most conveniently expressed in languages with rich support for both ETL and ML tasks, such as Python or R, but these implementations do not scale. In enterprise setups, the source data usually resides in a data warehouse. One possible strategy in such situations is to run the ETL part of the pipeline in situ, and the ML part in a specialized engine such as SciDB [11] or RasDaMan [6]. This approach has two drawbacks. First, moving data between engines is an expensive operation that is frequently repeated as the pipeline is refined. Second, it does not allow to easily join warehouse and external data sources.

Parallel dataflow engines such as Spark [38] or Hadoop [5] offer a more flexible execution infrastructure that does not suffer from the problems outlined above. Initially developed for ETL-like workloads, these systems have been increasingly used by practitioners to implement ML algorithms [26, 8, 33]. To support scalable execution of ML workloads, the functionality of established libraries for scalable linear algebra, such as ScaLAPACK [13], is being implemented on top of parallel dataflow systems by projects like SystemML [17], MLlib [26], Apache Mahout Samsara [33] and Pegasus [19].

A common runtime engine avoids data transfer, but the mismatch in data representation still manifests itself when executing mixed analytics pipelines. While dataflow engines typically row-partition large datasets, scalable linear algebra operators are implemented on top of block-partitioned, or *blocked* matrices. The difference in the partitioning assumptions results in a re-partitioning barrier whenever a linear algebra operator follows a relational one. The dataflow engine has to re-partition the entire row-partitioned dataset into a block-partitioned matrix. One possible solution would be to execute linear algebra operators on row-partitioned

Proceedings of the VLDB Endowment, Vol. 10, No. 13

matrices. Although this performs well for operations such as row sums (shown in Figure 1), superlinear operations such as matrix multiplication that consume multiple rows and/or columns become very inefficient [17]. For computational and storage efficiency, the majority of scalable linear algebra frameworks perform matrix multiplications on blocked matrices (e.g., [17, 18, 19]).

In this paper, we demonstrate the optimization potential of fusing relational and linear algebra operators. As a first step, we focus on a common pattern – a relational join, followed by a per-element transformation for feature extraction and vectorization, and a subsequent matrix conversion. To reduce the total shuffling costs of this operator chain, we propose *BlockJoin*, a specialized distributed join algorithm that consumes row-partitioned relational data and directly produces a block-partitioned matrix. We focus on the major drawback posed by an independent operator chain: The intermediate result of the join, row-wise partitioned by the join key, is discarded immediately to form a block-partitioned matrix. This materialization implies the risk of running out of memory when the join result becomes large, and more importantly results in a unnecessary shuffle operation for the join. BlockJoin avoids the materialization of the intermediate join result by applying the vectorization function and the successive block partitioning independently to both relations. Analogous to joins that have been proposed for columnar databases [24, 9, 1], BlockJoin builds on two main concepts: *index joins* and *late materialization*. More specifically, we first identify the matching tuple pairs and their corresponding row indexes in the matrix by performing a join on the keys and tuple-ids of the two relations (analogous to TID-Joins [25]). Based on the gathered metadata, we apply the vectorization function separately to the matching tuples of both relations, and repeat this for the block partitioning, without having to materialize the intermediate join result. Therefore, we can apply different materialization strategies for the matrix blocks based on the shape of the input relations, namely *Early* and *Late* materialization. Our experiments show that BlockJoin performs up to 6× faster than the state-of-the-art approach of conducting a row-wise join followed by a block-partitioning step.

Overall, we make the following contributions:

- We demonstrate the need for implementing relational operators producing block-partitioned datasets (Section 2.2).
- We propose BlockJoin, a distributed join algorithm which produces block-partitioned results for workloads mixing linear and relational algebra operations. To the best of our knowledge, this is the first work proposing a relational operator for block-partitioned results (Section 3).
- We provide a reference implementation of BlockJoin based on Apache Spark [38] with two different block materialization strategies (Sections 4).
- We provide a cost model to select the best suited materialization strategy based on the shape of the input tables (Section 3.4).
- We experimentally show that BlockJoin outperforms the baseline approach in all scenarios and, depending on the size and shape of the input relations, is up to 6× faster. Moreover, we show that BlockJoin is skew resistant and scales gracefully in situations when the state-of-the-art approach fails (Section 5).

2. BACKGROUND

In this section, we introduce the blocked matrix representation. We also discuss a running example we will use throughout the paper and discuss the state-of-the-art implementation for dataflow systems.

2.1 Block-Partitioned Matrix Representation

Distributed dataflow systems use an element-at-a-time processing model in which an element typically represents a line in a text file or a tuple of a relation. Systems that implement matrices in this model can choose among a variety of partitioning schemes (e.g., cell-, row-, or column-wise) for the matrix. For common operations such as matrix multiplications, all of these representations incur huge performance overheads [17]. Block-partitioning the matrix provides significant performance benefits. This includes a reduction in the number of tuples required to represent and process a matrix, block-level compression, and the optimization of operations like multiplication on a block-level basis. These benefits have led to the widespread adoption of block-partitioned matrices in parallel data processing platforms [17, 18, 19]. A blocked representation splits the matrix into submatrices of fixed size, called *blocks*. These blocks become the processing elements in the dataflow system.

2.2 Motivating Example

Our running example is learning a spam detection model, a common use case in e-commerce applications. Assume that customers write reviews for products, some of which are spam, and we want to train a classifier to automatically detect the spam reviews. The data for products and reviews are stored in different files in a distributed filesystem. We need the attributes from both relations to build the features for the model in our ML algorithm. Therefore, we first need to join the records from these tables to obtain reviews with their corresponding products. Next, we need to transform these product-review pairs into a suitable representation for an ML algorithm. To this end, we apply a user defined function (UDF) that transforms the attributes into a vector representation. Finally, we aggregate these vectors into a distributed, blocked feature matrix to feed them into an ML system (such as SystemML).

Figure 2 illustrates how to execute such a workload. Listing 1 shows how it can be implemented in a distributed dataflow system like Spark, expressing a mixed linear- and relational-algebra pipeline. We will refer to this as *baseline* implementation in the rest of the paper. The input data resides in the tables `Products (product_no, name, price, category)` and `Reviews (product_no, text, num_stars, is_spam)`. Step ❶ (in Figure 2 and Listing 1) performs a foreign-key join on the `product_no` attribute. Step ❷ applies user-defined vectorization functions to each row of the join result, to transform it into vector-based features, using techniques like feature hashing and "one-hot-encoding".

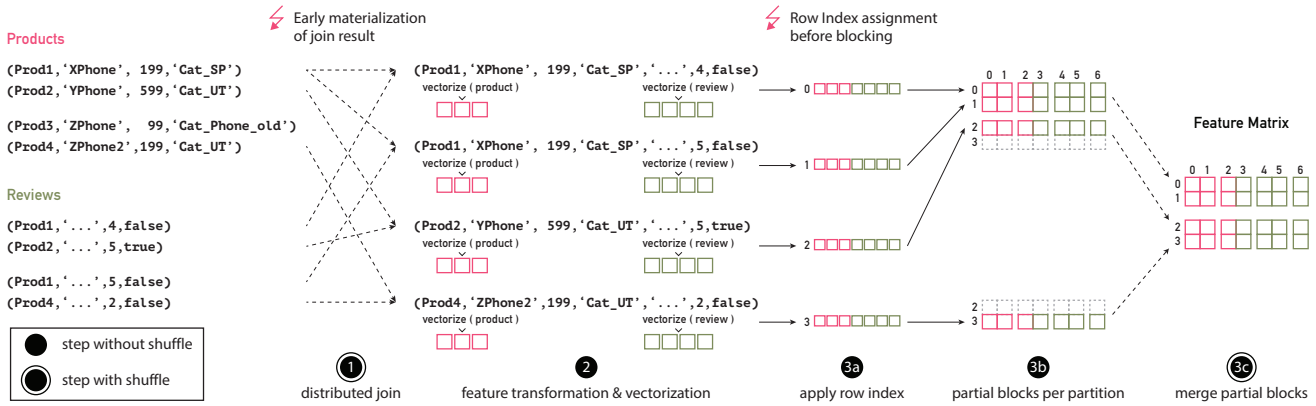


Figure 2: The baseline implementation for our running example. We prepare the data in order to learn a spam classifier in an e-commerce use case: ❶ we perform a distributed join of Products and Reviews, ❷ call user code to transform the join result into feature vectors, and ❸ (a) assign consecutive rows indexes to the resulting feature vectors, (b) create partial blocks per partition, and (c) merge them into a blocked matrix.

We assume that the vector resulting from a row is a concatenation of the vectorization of the input tuples of the participating relations. Step ❸ is split into three sub-steps that are necessary to form a block-partitioned matrix: (*a*) creates a sequential index for the join result that is used as row index for the matrix. This is necessary, as dataflow engines, in contrast to database systems, do not provide a unique tuple identifier. (*b*) builds the initial matrix blocks by splitting the rows at block boundaries. (*c*) in a final aggregation step, where partially filled blocks (which span multiple data partitions) are merged.

```
1   val Products: Dataset[Product] = // read csv...
2   val Reviews:  Dataset[Review]  = // read csv...
3
4 ❶ val JoinResult = Products.joinWith(Reviews,
5     Products("product_no") === Reviews("product_no"))
6
7   // Vectorize each tuple in the join result
8 ❷ val Vectorized = JoinResult.map { case (p, r) =>
9       val pv = vectorizeProduct(p)
10      val rv = vectorizeReview(r)
11      pv ++ rv
12  }
13
14  // Convert 'Vectorized' into blocked matrix 'M'
15 ❸ val M = toMatrix(Vectorized)
16  // Train the ML model with matrix 'M' ...
```

Listing 1: Code snippet for the running example.

3. BLOCKING THROUGH JOINS

In this section, we present BlockJoin, our chained, context-aware operator, leveraging the example of Figure 2. We first introduce a baseline implementation of independent operators for that example, which cannot leverage join metadata for the blocking phase. We then detail BlockJoin in Sections 3.1 and 3.2, and discuss how BlockJoin improves upon the baseline.

Drawbacks of an independent operator chain. The baseline implementation, which uses independent operators, is illustrated in Figure 2 and proceeds as follows: We first partition `Products p` by its primary key `p.product_no` and `Reviews r` by its foreign-key `r.product_no` to execute the distributed join. After vectorizing the join result `Vectorized v`, we introduce a consecutive index (e.g., by a *zipWithIndex* method in Spark), called *row-idx*, to uniquely identify each tuple. Then, we split each `v` of `Vectorized` into its components, based on the *col-idx*, and re-partition by the block index of the resulting matrix. The block index is obtained by the function:

$$block\text{-}idx(v, col\text{-}idx) = \{\frac{v.\textbf{row-idx}}{\textbf{block\_size}}, \frac{\textbf{col-idx}}{\textbf{block\_size}}\}$$

The `block_size` represents the number of rows and columns in a block. Although matrix blocks can have arbitrary row- or column-sizes, we use square blocks, for the sake of simplicity. One can easily derive the function for non-square blocks by substituting `block_size` with the number of rows and column per block.

We observe that an independent operator chain has to re-partition the data twice and materializes the join result, even though this result is split according to block boundaries immediately after applying the index assignment in Step 3a, as described before. Thus, the costly join is only executed to create a sequential index for the rows of the matching tuples in the matrix. Another danger during materialization of the join result is that the two input tables can be very wide, and we therefore risk running out of memory when executing the join.

In the following, we introduce BlockJoin and explain how it avoids to materialize the intermediate join result by introducing information exchange between the operators. We start by discussing a simplified case in Section 3.1, and extend our solution to the general case in Section 3.2.

3.1 BlockJoin under Simplifying Assumptions

We introduce two simplifying assumptions to explain how to independently block two relations[1]: (*i*) the join keys on both relations are *consecutive* integers and the relations are

[1]Note that we introduce these assumptions solely for the purpose of discussing the blocking, we drop these assumptions in the next section and describe how to apply BlockJoin for general equi-join cases.

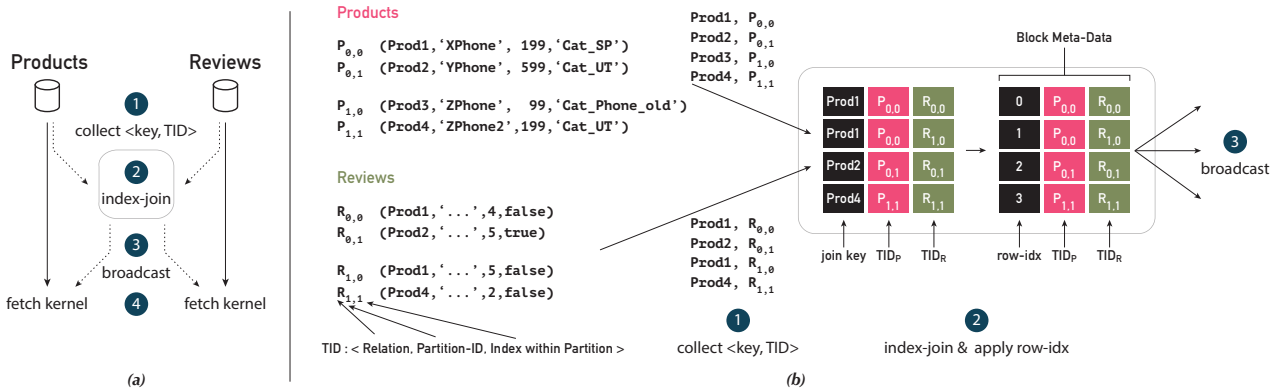


Figure 3: Local index-join & sequential row index assignment for the running example: (1) we collect the <key, TID> pairs on the join coordinator, (2) we perform an index-join on the collected tuples and introduce the sequential row index row-idx on the result. Afterwards, (3) we broadcast the result back to the nodes. The fetch-kernel (4), is shown in Figure 4.

ordered by their keys; (*ii*) there is a strict 1:1 relation between the tables, that is: they have the *same cardinality* and the *same values* in their primary key. Joining two relations, which fulfill these conditions, is equivalent to *concatenating* the relations. Moreover, the cardinality of the join result will be the same as the cardinality of the two joined relations. Now, suppose that we want to block-partition the join result of the two relations. The question we are going to answer throughout the rest of this section is: *Can we achieve joined and block-partitioned results, without first materializing the join result in a row-partitioned representation?*

Blocking without materializing the join result. Given our simplifying assumptions, we can safely treat the key product_no as the unique, sequential identifier of each tuple. Hence, we can not only use it as join key, but but can also define $v.\textbf{\textit{row-idx}} = v.\textbf{\textit{product\_no}}$, to uniquely identify the rows in the resulting matrix. Now, as we do not need to materialize the join result to obtain the *row-idx*, we discuss how we apply the blocking function on both relations independently after the vectorization. The first component of the *block-idx* function ($\frac{v.\textbf{\textit{row-idx}}}{\textbf{block\_size}}$) assigns the row index of the block *blk-row-idx*, which the cells in a row belong to. Due to our assumptions, matching tuples already share the same *row-idx*. The second component of the *block-idx* function ($\frac{v.\textbf{\textit{col-idx}}}{\textbf{block\_size}}$) defines the column index of the block *blk-col-idx*, which the cells of a rows are split across. We can use this part of the equation on the individual tables without joining after we apply some small changes: the function has to account for the fact that the *blk-col-idx* of the second relation have to be offset by the number of columns in the first relation (because the result concatenates the two relations). Thus, we add the offset cols(pv) (i.e., the number of columns of the vectorized version of the first relation p)[2] to the column index of the second relation. Equation 1 shows the modified *block-idx* function that is applied on the vectorized tuples of the individual input relations.

$$\begin{aligned} \textit{block-idx}_{\boldsymbol{P}}(pv, \textit{col-idx}) &= \{\tfrac{pv.\textbf{\textit{row-idx}}}{\textbf{block\_size}}, \tfrac{\textbf{\textit{col-idx}}}{\textbf{block\_size}}\} \\ \textit{block-idx}_{\boldsymbol{R}}(rv, \textit{col-idx}) &= \{\tfrac{rv.\textbf{\textit{row-idx}}}{\textbf{block\_size}}, \tfrac{\textbf{cols(pv)} + \textbf{\textit{col-idx}}}{\textbf{block\_size}}\} \end{aligned} \quad (1)$$

[2]Section 4 details how we determine this value at runtime.

3.2 BlockJoin for the General Case

The simplifying assumption of an ordered, consecutive index on both relations from the previous section obviously does not hold in reality. In real-world scenarios, we observe *primary-key* (PK) – *foreign-key* (FK) or 1:N relationships, such as users and items, items and reviews, or even M:N relations, as well as normalized database schemata [21]. Therefore, we cannot use the keys of the individual relations to determine the corresponding blocks of the tuples. Moreover, the size of the input relations may vary compared to the join result. For instance, a Product can match arbitrarily many Reviews. In the subsequent paragraphs, we show how BlockJoin determines which tuple pairs are part of the join result, and assigns a unique *row-idx* to each matching tuple pair under general conditions without materialization of the join result.

Assigning indexes to tuple pairs in the join result. BlockJoin first obtains a unique surrogate key *TID* from each tuple of both relations independently. The *TID* consists of a <relation-id, partition-id, index-within-partition> triple as depicted in the bottom left part of Figure 3 (b). The triple uniquely identifies each row of the relations. In the next step, we generate the unique identifier *row-idx* for the rows in the *resulting* Matrix M. In order to assign the identifier to the matching tuples of both relations, we design a variant of the index-join [15, 25]. The main idea of the index-join is to project the key and TID columns of the two relations to determine matching tuples without materializing the *payload* columns. As depicted in Figure 3, step (1) projects and collects the <key, TID> pairs from both relations on the driver. Therefore, we have all keys of the two relations and execute an *index-join* (2). Based on the result, we assign the *row-idx* to the matching tuples. We call this phase *join-kernel*, following the nomenclature of [36]. In Step (3), we make the *block metadata*, which contains the matched <key, TID> pairs and *row-idx*'s, available on all nodes for the subsequent *fetch-kernel* phase. Based on the information in the metadata, we prune all non-matching tuples and apply the vectorization function to the remaining tuples (4) on each relation separately. While we can use the very same *block-idx* function, as described in Section 3.1, Equation 1, we elaborate on two different strategies for efficient blocking, enabled by applying the *row-idx* separately, in the next section.

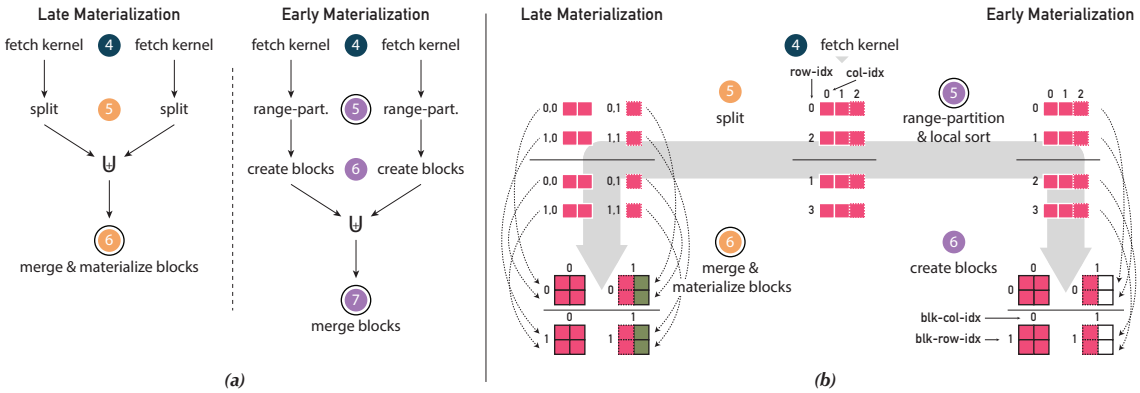


Figure 4: Block-materialization strategies. We illustrate the materialization strategies for one relation (Products) in Figure b. *Late materialization* breaks the tuples into multiple row-splits locally 5 and merges the splits of both relations after union them 6. *Early materialization* first range partitions the complete rows in order to group tuples belonging to the same blocks and then performs a local sort on the row index to enable faster block creation 5. After materializing the blocks per relations 6, potentially partial blocks are merged.

3.3 Block Materialization Strategies

Figure 4 (a) sketches the two materialization strategies for BlockJoin. Both approaches share the initial Steps 1 to 3 from Figure 3, explained in the previous section. The main difference stems from the *block materialization* strategy we use for the values emitted in Step 4, the *fetch-kernel*.

Our goal now is to shuffle the row-splits[3] of each row to the nodes responsible for the splits' destination blocks. A very important consideration is that one row-split may need to fill multiple rows in the same block and might be part of multiple blocks. For instance, consider a row-split of a product which matches multiple reviews. If there are 10 matches and the `block_size` is 5, that product's row-split will have to be duplicated 10 times and, therefore, contribute to at least 2 different blocks. Duplicates can have a huge impact on the runtime of the block materialization phase. For this reason, we devise two materialization strategies which are detailed below.

Late Materialization. The left side of Figure 4 (b) depicts the execution flow of late materialization. The key idea behind late materialization is to reduce the number of row-splits emitted, by sending each split only *once per destination block*, even if the row-split occurs multiple times in the respective block. The duplicates of each split are materialized on the receiver side for each block. We can apply receiver-side materialization, as we are not forced to materialize the join result (like in the baseline), to obtain the sequential *row-idx*. More specifically, each row emitted from the fetch kernel 4 is split in multiple `<blk-idx, row-offset, duplicates, row-split>` tuples 5. Since there might be multiple matches for a key, we store the number of `duplicates` per block, instead of materializing them early. The `row-offset` defines the first row index of the row-split in the destination block. In the destination node, we merge the row-splits of the same `blk-idx`, and create the complete blocks by materializing their duplicates 6. Note that we create complete blocks even in the case they contain data from both relations in one pass (as can be seen for the green cells from the Reviews table).

Early Materialization. The right side of Figure 4 (b) depicts the execution flow of early materialization. Instead of separating the rows from the fetch kernel 4 into row splits immediately, we emit *a single* `<row-idx, duplicates, row>` tuple per row. Rows matching multiple times are not yet materialized, and we emit one tuple for all duplicates within a block again. In the next step, we *range-partition* the tuples by their `row-idx` and sort within each partition 5. A custom partitioner ensures that tuples belonging to the same block end up in the same partition. Next, we create the blocks and materialize the duplicates for each relation separately 6. Note that we do not have to shuffle, but potentially create partial blocks (as can be seen for the blocks with column index 1). In the last step, we `union` the relations and merge the partial blocks.

Applicability to the baseline. While we can apply the presented materialization strategies also in the baseline, we do not gain any advantage. The main benefit of late materialization is the receiver-side materialization of duplicates (e.g., PK matching multiple FKs). In the baseline though, we materialize all duplicates during the distributed join phase. As a result, we shuffle the same amount of data as in the baseline, but with a much larger amount of tuples, as we split the rows in late materialization. The advantage of early materialization yields from the custom partitioner, which ensures partitions that do not span over block boundaries. In BlockJoin, we introduce the shuffle needed for this partitioner, as we do not shuffle for the distributed join that is required in the baseline. Therefore, applying the partitioner on the baseline would introduce an additional shuffle step, making it worse than the baseline.

3.4 Choosing a Materialization Strategy

To make these trade-offs between late and early materialization more concrete, we compare the two materialization strategies against the baseline implementation described in Section 2.2. We base our comparison on the cost model shown below, using the symbols from Table 1. For brevity and simplicity, we focus only on the amount of data ex-

[3]Given a row r, a row-split is a tuple which contains a strict subset of the columns or r. The purpose of a row-split is to fit in a given block. For instance, given a block size of 2, a row with 6 columns will be split into 3 row-splits.

Table 1: Cost model notation.

Symbol	Meaning
$\|T\|$	Number of rows in relation T
$cols(T)$	Number of columns in relation T
$bytes(T)$	Size (bytes) of a tuple in relation T
$\mathbf{b}$	Number of rows/columns per square block
P, R	Input tables of the join
J	Join result

change and the number of tuples during the shuffling phases, and make the simplifying assumption that all the tuples of the two input relations survive the join, which also reflects the worst case for our materialization strategies. Late materialization emits multiple row-splits per row, thus increases the number of tuples to be shuffled. On the other hand, early materialization emits full (and materialized) blocks at the expense of an extra range-partitioning on complete rows and local sorting step. Since the blocks in the early materialization schema are complete, apart from blocks containing columns from both relations (which is equal to number of row-wise blocks), only those have to be considered during the merging process.

Size of Shuffled Data.

$$\begin{aligned}
&baseline && \rightarrow |P| \cdot bytes(P) + |R| \cdot bytes(R) && join\\
&&& + |J| \cdot bytes(J) && merge\ blocks\\
&early && \rightarrow |P| \cdot bytes(P) + |R| \cdot bytes(R) && range\text{-}partition\\
&&& + |J| \cdot bytes(J) && merge\ blocks\\
&late && \rightarrow |P| \cdot bytes(P) + |R| \cdot bytes(R) && merge\ blocks
\end{aligned}$$

Deriving the size of shuffled data for the baseline implementation is straightforward: we execute a shuffle in order to perform the join ($|P| \cdot bytes(P) + |R| \cdot bytes(R)$) and another shuffle of the join results for block-partitioning them ($|J| \cdot bytes(J)$). The early materialization strategy has to shuffle the input data in order to range-partition it ($|P| \cdot bytes(P) + |R| \cdot bytes(R)$) and shuffle the join result in order to merge the blocks ($|J| \cdot bytes(J)$), as we might have partially filled blocks. Finally, the late materialization strategy only needs to shuffle once to merge all row-splits in their corresponding block ($|P| \cdot bytes(P) + |R| \cdot bytes(R)$). The late materialization strategy is expected to have the least amount of data shuffling. However, the amount of tuples exchanged differs among the three implementations.

Number of Shuffled Tuples.

$$\begin{aligned}
&baseline && \rightarrow |P| + |R| && join\\
&&& + \tfrac{|J|}{\mathbf{b}} \cdot \tfrac{cols(J)}{\mathbf{b}} && merge\ blocks\\
&early && \rightarrow |P| + |R| && range\text{-}partition\\
&&& + (\tfrac{|J|}{\mathbf{b}} \cdot \tfrac{cols(J)}{\mathbf{b}}) + \tfrac{|J|}{\mathbf{b}} && merge\ blocks\\
&late && \rightarrow |J| \cdot \tfrac{cols(J)}{\mathbf{b}} && merge\ blocks
\end{aligned}$$

The number of tuples exchanged for the baseline implementation includes the relations themselves ($|P|+|R|$), plus the total number of blocks that form the final matrix. The number of blocks is defined by the rows in the join result divided by the block size ($\frac{|J|}{\mathbf{b}}$) and the number columns, divided by the block-size ($\frac{cols(J)}{\mathbf{b}}$). The early materialization strategy will require an extra $\frac{|J|}{\mathbf{b}}$ for the partial blocks that span both relations (detailed in the Block Materialization paragraph of Section 4). In the late materialization strategy, we emit each matching row of both relations ($|J|$) multiplied by the number of splits per row ($\frac{cols(J)}{\mathbf{b}}$). Intuitively, late materialization always emits more tuples than early materialization and the baseline, because each row of the result is split while the early materialization creates (partial) blocks before shuffling.

Estimating Cost. Estimating the runtime of BlockJoin, boils down to estimating a cost function which takes into account the amount of shuffled data as well as the number of shuffled tuples as we have shown earlier in this section. A straightforward implementation of a cost estimation function would simply calculate a linear combination of size and number of tuples and yield an estimated cost. To this end, one can train two regression models based on our previously presented formulas for computing data size and number of shuffled tuples for both materialization strategies. For early materialization, the regression $r_e = \begin{bmatrix} d_e(\theta) & t_e(\theta) & 1 \end{bmatrix}^\top \mathbf{w}_e$ predicts the runtime r_e. Here θ denotes a vector that contains the data statistics from Table 1 for a particular join input, $d_e(\theta)$ and $t_e(\theta)$ refer to the previously presented functions for computing data size and number of shuffled tuples for early materialization and $\mathbf{w}_e$ denotes the learned regression coefficients. Analogously, a regression model $r_l = \begin{bmatrix} d_l(\theta) & t_l(\theta) & 1 \end{bmatrix}^\top \mathbf{w}_l$ can be trained for predicting the runtime r_l for late materialization. The obtained regression coefficients depend on the actual cluster settings. Therefore, a couple of experiments must be executed to obtain a sample of different runtimes for different data characteristics, before the model can be fitted. Afterwards, the prediction model can be used to select the best suited materialization strategy for subsequent runs. We present such an instance of a trained model in our experiments and showcase its accuracy.

Using this model requires statistics on the input tables and the join result. We can integrate the model into an optimizer, which creates an optimized plan statically before job execution (e.g., Catalyst in Spark), but have to rely on table statistics and estimations for the join result to select the best strategy.

3.5 Extensibility

So far we have only considered equality-joins. However, BlockJoin and the general idea of assigning unique identifiers without materializing the intermediate join result is independent of the actual join algorithm that runs locally. Thus, extending BlockJoin for theta and n-ary joins boils down to implementing a variation of the index-join used to define the matching tuples. Theta joins can be implemented by a projection of the columns required for predicate evaluation and a modified version of the shared metadata, to identify matching tuples and conduct row index assignment in the fetch-kernel. Extending BlockJoin to n-ary joins is also possible, once we identify the join results. However,

this extension requires further research regarding the choice between multiple binary joins or a solution based on multi-way join algorithms, which we leave to future work.

4. IMPLEMENTATION ASPECTS

In this section, we present important technical aspects to consider when implementing BlockJoin in distributed dataflow systems.

Row Index Assignment. In order to block partition the join result, we need to assign consecutive row indexes to the join result. In the baseline implementation, we conduct this assignment on the distributed join result. For that, we leverage Spark's `zipWithIndex` operation, which counts the number of elements of each partition in the distributed dataset, and uses the result to assign consecutive indexes in a second pass over the data. In BlockJoin, we create the unique row indexes during the join-kernel based on the matching tuples and make them available as part of the metadata. Therefore, the assignment of row indexes to emitted tuples in the fetch-kernel phase can be done on each relation individually, without prior materialization of the join result.

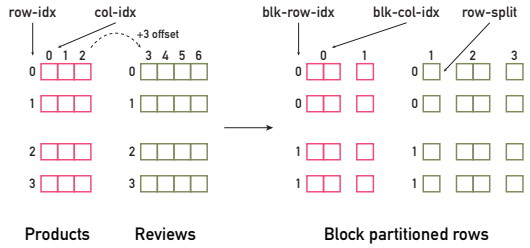


Figure 5: Tuples resulting from range-partitioning the vectorized tuples of `Products` and `Reviews` with block size 2×2.

Block Materialization. In the baseline implementation, we create the blocks after assigning the row index. To reduce the number of emitted partial blocks, the baseline uses a `mapPartitions` function to create the matrix blocks. This function provides an iterator over the whole partition inside the UDF. Due to the sequential row index, all rows that belong to a certain block come one after the other, which allows us to create full blocks before emitting. Therefore, we only have to combine blocks that are split row-wise between two partitions in the succeeding merge step.

As discussed in Section 3, we create the correct `block-idx` separately on both tables in the BlockJoin. Figure 5 shows the assignment of the block index in detail. We create partial blocks for the *blk-col-idx* 1 in both relations, as the block is split across both relations. In the late materialization approach, we have to merge all individual tuples on the receiver-side, which reduces the data that needs to be shuffled but increases the number of tuples in certain scenarios (as discussed in Section 3.3). In the early materialization approach, we also use a `mapPartitions` function to create full blocks on the sender-side. As we can not guarantee sorted row indexes for at least one of the relations, we would risk emitting partially filled blocks, as consecutive tuples might belong to different blocks. Therefore, we provide a custom partitioner, which creates partitions that do not cross block boundaries. Afterwards, we sort by the row index within each partition to create consecutive blocks. Thus, we only have to merge blocks that contain columns from both relations, e.g., for blocks with column *blk-col-idx* 1 in Figure 5.

Table 2: Size of dense data in GB.

Cols \ Rows	10K	100K	200K	500K	1M
1K	0.2	1.7	3.4	8.5	16.9
5K	0.9	8.5	16.9	42.3	84.7
10K	1.7	16.9	33.9	84.7	169.3
25K	4.2	42.3	84.6	211.6	423.2
50K	8.5	84.7	169.3	423.3	846.7
100K	16.9	169.3	338.7	846.7	1700.0

Determining Matrix Dimensions. In order to assign the vectorized data to matrix blocks, it is necessary to know the dimensionality of the vectors returned by the user-defined vectorization functions upfront. One can either require the user to specify this in the vectorization functions, or alternatively fetch a single random tuple from each relation once, apply the vectorization function, and record the dimensionality of the resulting vector.

5. EXPERIMENTAL EVALUATION

In this section, we comprehensively evaluate experiments comparing BlockJoin with late and early materialization against a baseline approach on dense and sparse data. As discussed before, the baseline represents the current state-of-the-art: we use Spark to execute the join of the tables, and then SystemML to create a blocked matrix representation from the join result without staging the intermediate results on HDFS.

Sparsity mainly affects the data size and runtime, but not the overall performance trend for the algorithms. For this reason, we show the results for sparse and dense data for each experiment in the same plot. Throughout the experiments, sparse data is indicated with *patched* bars in the front, whereas dense data is indicated with *solid* bars.

Setup. We used a local cluster with up to 20 worker nodes connected with 1 GBit Ethernet connections. Each machine is equipped with a quad-core Intel Xeon X3450 2.67 GHz, and 16GB of RAM. We implemented BlockJoin on Spark 1.6.2 (each Spark worker has 4 slots) and store the initial data in HDFS 2.4.1. Every experiment is executed seven times and we report the median execution time. For the experiments on dense data we use 20 worker nodes, resulting in a degree of parallelism (DOP) of 80, while we use 10 worker nodes (DOP = 40) for sparse data.

Dataset. In order to have full control of the shape, size and content of the input tables we evaluate BlockJoin on synthetic datasets. The simulated tables, called PK and FK, have following schema: PK (`key,` r_1`, ...,` r_n) and FK (`fKey,` s_1`, ...,` s_m). We use a vectorization function that converts r_1, ..., r_n to an n-dimensional double precision vector, and analogously s_1, ..., s_m to an m-dimensional double precision vector. We conducted the experiments for dense and sparse (10% non zero values) vectors and vary the number of rows and columns. If not stated otherwise in the experiments, the tables have a 1:N primary key - foreign key relation. We use squared blocks of 1000 × 1000 as it

was shown to make a good trade off between computational efficiency and network traffic [17]. The corresponding sizes of the tables are given in Table 2.

In addition, we provide experiments on the publicly available *Reddit Comments*[4] dataset. It consists of line separated JSON entries that represent comments on the news aggregator website *Reddit*. Each JSON entry contains a single comment with additional information such as the author, votes, category, etc. We split the raw data into a comment and author CSV file, by introducing a primary - foreign key relation *author_id* and use these as input to our experiments. The final join input are ∼30 million comments (5.1 GB) and ∼1.5 million authors (29.9 MB).

Data Distribution. Many real-world datasets exhibit extreme skew in the distribution of data points per object observed (e.g., reviews per product), and it has been shown that this skew increases over time in many datasets [23]. When joining with such datasets, a small number of tuples from the skewed relation will produce a very large amount of tuples in the join result. For this reason, we conduct experiments with uniform as well as power-law distributed foreign keys (with $\alpha = 0.01$).

5.1 Effect of Table Shape and Size

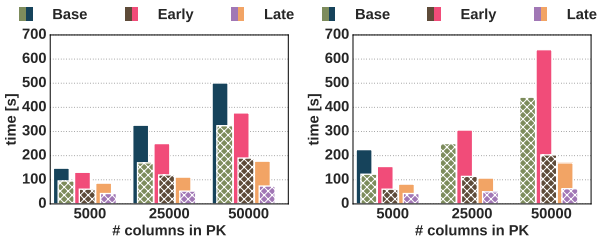
In this experiment, we evaluate the scalability of BlockJoin for different numbers of columns. We fix the rows to 100K in the PK and 1M in the FK table. All rows in the FK table match at least one key in the PK table. Therefore, we concentrate on the effects of the block materialization strategies, as BlockJoin can not gain performance by pruning non-matching tuples (an expected effect of the *fetch-kernel* phase).

Scaling PK Columns. In this experiment we fix the number of columns in the FK table to 5K, while we scale the PK table, from 5K to 50K columns, until it reaches the same data size as the FK table.

Figure 6 (a) depicts the results for uniform distributed foreign keys. A first observation is that Late Materialization scales much better and is up 2.5× faster than the baseline for sparse and dense data. Late Materialization materializes duplicates (primary keys matching multiple foreign keys) at the receiver side. Thus, it only needs to shuffle data equal to the size of the input tables. In contrast, both Early Materialization and the baseline approach, materialize the duplicates (the baseline approach in the join and Early Materialization before merging partial matrix blocks). Therefore, they shuffle up to 847GB + 84,7GB (for 50K dense columns); roughly 10× more data compared to Late Materialization. Even though the baseline and Early Materialization shuffle the same amount of data, Early Materialization appears to outperform the baseline by 10%. The faster execution of Early Materialization is due to *(i)* the independent blocking of the two relations without materializing the join result, and *(ii)* our custom partitioner (see Section 4), which never splits rows sharing the same `blk-row-idx` across different partitions.

Figure 6 (b) shows the same experiment for power-law distributed foreign keys. Note that the baseline approach fails to perform the join for more than 5K columns of dense data. We experienced an internal Spark error, while it tried to read partitions to execute the join on the receiver side.

[4]http://files.pushshift.io/reddit/comments/

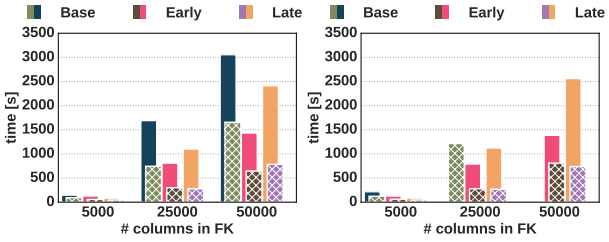


(a) Uniform (b) Power-law

Figure 6: Effect of scaling the number of columns in the PK table. The number of FK columns is fixed to 5K.

This is due to the heavily skewed data, which results in almost all of the work ending up in one worker node, which is unable to gather and sort the received partitions. For Late Materialization, we can observe that the algorithm is not affected by data skew and outperforms the baseline by up to 4× for sparse data. The effect of skewed keys on Early Materialization is not as severe as for the baseline, but the heavily increased amount of duplicates, still decreases its performance as the PK table holds the majority of the data.

Scaling FK Columns. Figure 7 (a) depicts the inverse experiment with 5K in the PK table and scaling number of columns in the FK table. This time, Early Materialization outperforms the Late Materialization for dense data and performs up to 2× better than the baseline. Note that in this experiment, (*i*) the FK table grows very large, up to 846.7GB for dense data, in comparison to the previous experiment, while (*ii*) the resulting matrix sizes are exactly same. Thus, as the PK table accounts for the duplicates, Late Materialization does not save much by late duplicate materialization. However, the number of shuffled FK tuples increases with the number of columns in the FK table. Late Materialization emits up to 50M (1M rows split in 50K columns divided by 1K block size) row-splits, while only 1M rows are exchanged by Early Materialization and the baseline.



(a) Uniform (b) Power-law

Figure 7: Effect of scaling the number of columns in the FK table. The number of PK columns is fixed to 5K.

Figure 7 (b) shows the experiment with a power-law distributed foreign keys. For the two versions of BlockJoin, we can observe almost the same runtime as for the uniform distributed keys, as the data size is dominated by the FK table. Therefore, the impact of the skewed keys on Early Materialization is minor and Late Materialization does not save much data exchange. This time, the baseline approach fails to finish the experiment in case of more than 25K sparse columns due to the increased size of the FK table.

Experiment Conclusion. When the PK table size dominates the data exchange, Late Materialization performs up to 4× better than the baseline and outperforms Early Materialization. However, when the FK table dominates data exchange and the duplication of row-splits is no longer an issue, Early Materialization can be up to 1.8× faster than Late Materialization and 2× faster than the baseline. Finally, we were unable to conduct all experiments for the baseline in case of skewed data and the performance of Late Materialization is generally less affected by the data distribution.

Cost Model Evaluation. We trained the regression models, described in Section 3.3, based on the experiment results using dense input data. Figure 8 depicts the estimated runtime in relation to the number of columns in the two input relations. The number of rows is thereby the same as in the experiments (100K for PK and 1M for FK). We can observe that the model reflects the measured runtimes. While the model can serve as binary classifier to select the best suited strategy for other experiments, we are aware that we need more data to fit the model thoroughly. Another interesting observation is that we can use the column distribution as a simplified measure to select the strategies ($cols(PK) > cols(FK)$ favors Late Materialization and vise versa). This ratio turns out to be a pretty good estimation model and can be used as a fallback in an optimizer, as long as not enough training data is available to fit the model.

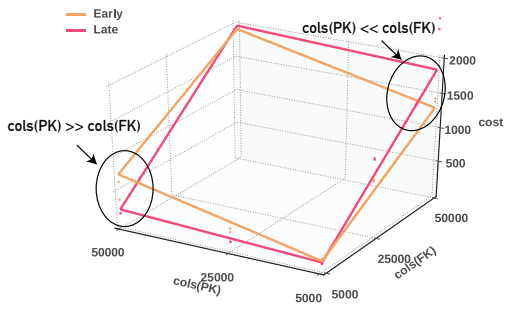


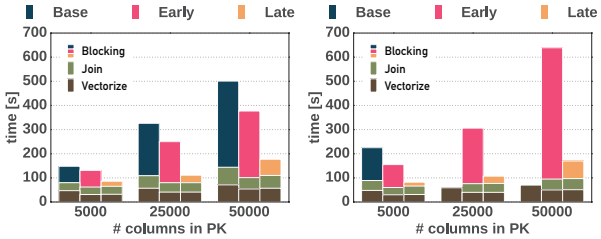
Figure 8: Estimated cost of the regression models, trained on the experiment results from Section 5.1. The number of rows correspond to the experiments. The data points represent the experiment results for Late Materialization and Early Materialization.

Detailed runtimes of the different phases. In Figure 9 and 10, we show the runtime of each of the phases – vectorize, join, and blocking – for the experiments with dense data in Figure 6 and 7 respectively. Due to operator chaining in Spark, we had to measure the phases in separated jobs to obtain their individual runtime.

Vectorize – We observe roughly equal run times, which is expected, as the same vectorization function is performed for both the baseline and BlockJoin.

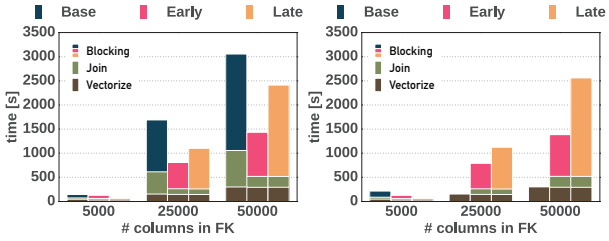
Join – We observe different behavior depending on whether we scale the PK or FK columns. Scaling the PK columns (Figure 9), we see only a minor speedup for BlockJoin in case of uniform distributed keys. For power-law distributed keys, the baseline fails to execute the join after 5K columns. As expected, BlockJoin is not sensitive to skewed keys and the join times are equal to the cases with uniformly distributed keys. Scaling the FK columns (Figure 10), we observe a speedup of up to 3x. Compared to Figure 9, we have to shuffle much more data, as we increase the FK columns. BlockJoin degrades gracefully with increasing number of columns, as we have to read the data to project the join keys. Again, the baseline fails to execute the join for power-law distributed keys, while BlockJoin is not affected by skew.

Blocking – We observe performance gains of up to 3x for the best suited materialization strategy. This applies mainly for late materialization, as the benefits are rather small in cases early materialization is better. The gains in performance for early materialization are due to the block-size aware partitioning. Late materialization gains performance due to the receiver-side materialization of duplicates. Thus, we observe a huge performance gain when scaling the PK columns. The behavior reflects the assumptions of our cost model: When scaling the PK columns, Late Materialization is superior as it avoids the materialization of the duplicates in the PK table and thus, shuffles considerably less data. When we scale the FK columns, Late Materialization can not gain much from receiver-side materialization as the majority of data resides in the FK table, but has to shuffle way more tuples. The experiments show that BlockJoin gains performance with both, a efficient, skew resistant join and the right choice of the materialization strategy.



(a) Uniform (b) Power-law

Figure 9: Split up execution times for scaling the number of columns in the PK table.



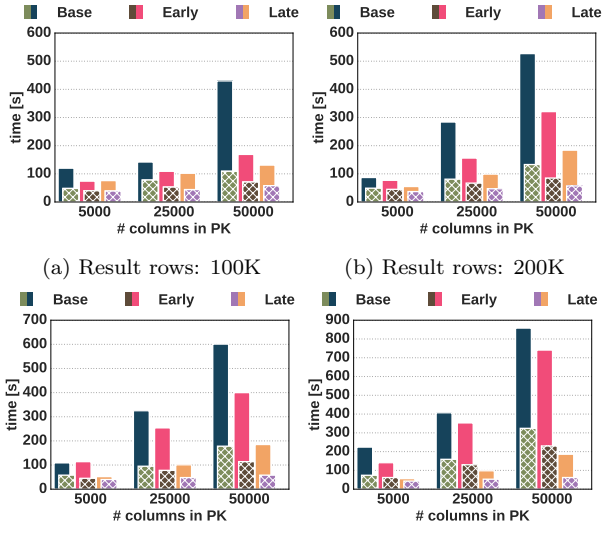
(a) Uniform (b) Power-law

Figure 10: Split up execution times for scaling the number of columns in the FK table.

5.2 1:1 and M:N Relations

In this experiment, we analyze the effects of 1:1 and M:N relations between the keys in the two relations. Therefore, we fix the number of rows in both tables to 100K and use sequential keys in both relations, but vary the range we draw the keys from. Figure 11 (a) depicts a 1:1 relation; each key appears once per table. Late Materialization and Early Materialization gain up to 2× speedup compared to the baseline

(both for sparse and dense data). As there are no duplicates, Early Materialization is only slightly slower than Late Materialization. Figure 11 (b) – (d) illustrate M:N relations with 2, 4, and 10 duplicates per key, and therefore, 200K, 400K, and 1M rows in the matrix. While the baseline has the worst performance throughout the series, we can observe a declining performance of Early Materialization with increasing number of duplicates for dense data. The runtime of Late Materialization is almost not affected by the number of duplicates and gains up to 4× speedup compared to the baseline for dense and sparse data.



(a) Result rows: 100K (b) Result rows: 200K

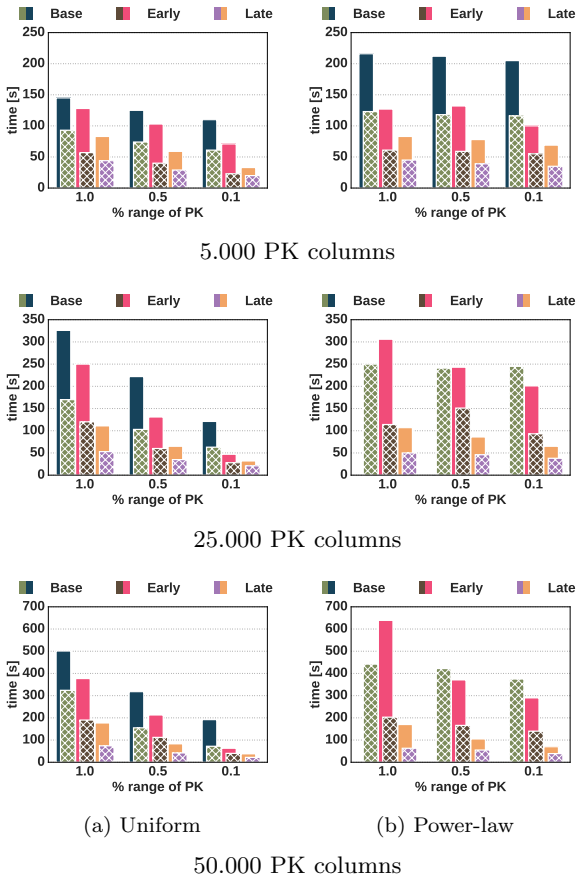
(c) Result rows: 400K (d) Result rows: 1M

Figure 11: Effect of scaling the number of columns in PK table for: (a) 1:1 relations and (b) – (d) M:N relations with 2, 4, and 10 duplicates per key.

5.3 Effect of Selectivity

In this experiment, we investigate the performance implications of the join selectivity. Therefore, we can observe the impact of the semi-join reduction in the fetch-kernel. We start with the same number of rows in the PK and FK table as in the previous experiment (Section 5.1), but we restrict the number of tuples in PK table. As a result, not all foreign keys match. This reflects a common use case, where only certain values, e.g., products of a given category, are of interest.

Scaling PK Columns. Figure 12 shows the experiment with fixed FK columns (5K) and scaling PK columns. On the x-axis, we increase the selectivity of the filter on the PK table. The selectivity not only defines the number of rows in the PK table (from 100K to 10K rows), but also the number of matching foreign keys, and thereby the size of the join result/matrix. Again, Late Materialization outperforms Early Materialization, but the benefits of late duplicate materialization decrease with increasing selectivity. Nevertheless, we achieve up to 4× speedups, due to pruning non-matching tuples in the fetch-kernel. For power-law distributed keys (Figure 12 (b)), the baseline approach fails for PK tables with more than 5K columns of dense data and the skew resistant Late Materialization gains up to 6× speedups for sparse data.



(a) Uniform (b) Power-law

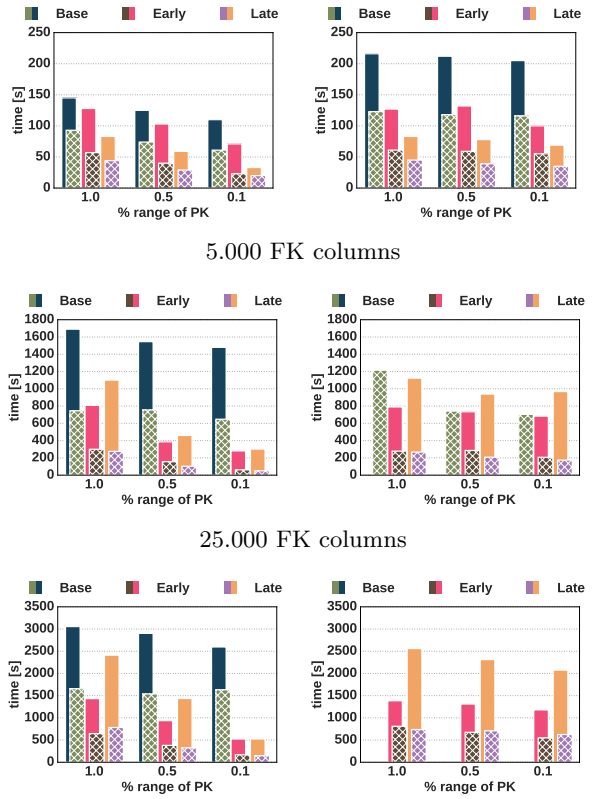
Figure 12: Effect of selectivity for varying number of columns in the PK table. The number of FK columns is fixed to 5K.

Scaling FK Columns. Figure 13 depicts the experiments with scaling number of columns in the FK table. Again, we can observe the performance degradation of Late Materialization, compared to the experiments in Figure 12, as the number of FK columns increases. Note that increasing selectivity mitigates the performance impact of row splitting for Late Materialization due to pruning in the the fetch-kernel and we see almost equal performance for Early Materialization and Late Materialization in case of 0.1 selectivity. The semi-join reduction thereby increases the speedups from 2× for 1.0 up to 6× for 0.1 selectivity. Figure 13 (b) shows the experiment with power-law distributed keys. While Late Materialization can outperform Early Materialization in the smallest configuration, pruning cannot mitigate the exploding number of tuples for larger number of columns in the dense case.

Experiment Conclusion. Restricting the primary key table to certain categories or values is a common use case. We showed that the impact of pruning in BlockJoin further increases its performance benefits compared to the baseline up to 6×.

5.4 Reddit Comments Dataset

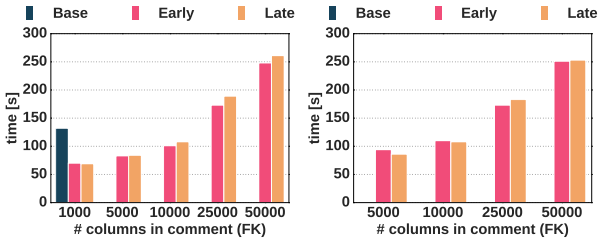
In this experiment, we evaluate our join algorithms on the Reddit Comments dataset, described in the beginning of this section. In order to obtain the full feature set, we



(a) Uniform (b) Power-law

Figure 13: Effect of selectivity for varying number of columns in the FK table. The number of PK columns is fixed to 5K.

join the comments and authors CSV input files. To create a vector representation, we apply *feature hashing* to the authors name and the comments text. We split the name by camel case, white space, and other delimiters and hash the words to a fixed size feature space. For the comments, we split the text into words and hash them as described before.



(a) User vector dim. = 1000 (b) User vector dim. = 5000

Figure 14: Effect of scaling the number of columns for the comment relation.

Figure 14 depicts the results of the experiment. We fix the dimensions of the author name feature vector to 1000 and 5000 and increase the dimensions of the comments vector. The first observation is that the baseline implementation fails after the first scaling factor. This is due to an out of memory exception in the blocking phase. The large amount of comments (∼30 million tuples) exceed the available memory in the `mapPartitions` operators that create partial blocks within each partition. While we also create partial blocks in the early materialization approach, we execute the blocking on the two relations separately, without prior joining. This leads to less memory pressure, compared to the baseline. Late materialization is not affected by memory pressure. This leads, in combination with the huge difference in the relations sizes (1 : 30) and the relatively small sparse feature vectors, to an almost equal runtime for Late Materialization and Early Materialization.

6. RELATED WORK

Join Optimization. Optimized join algorithms have been well studied in the area of distributed database systems [27, 31, 35, 30, 3] and parallel dataflow systems [29, 37, 2, 28, 40] like Hadoop [5] and Spark [38], with the aim of reducing network traffic and dealing with skewed data. Efficient join implementations in main-memory databases are based on TID-joins [25, 15] and late materialization [36, 24] to achieve cache efficiency up to the latest possible point. In BlockJoin, we apply and enhance these techniques for the domain of distributed matrix computation by using *index-joins* to create the matching tuples without re-partitioning the tables. More specifically, we apply a *semi-join* reduction to prune tuples before creating the blocks and we introduce *late materialization* to avoid sending rows resulting from duplicated join keys.

Array Databases. RasDaMan [6] is an array DBMS for multidimensional discrete data with an extended SQL query language. It stores its data as *tiles*, i.e., possibly non-aligned sub arrays, as blobs in an external DBMS. While their optimizer provides a rich set of heuristic-based rewrites, to the best of our knowledge, RasDaMan does not perform joint optimization over relational and array backed data. SciDB [11] is another array database that, in contrast to RasDaMan, provides its own shared-nothing storage layer. This allows SciDB to store and query tiles more efficiently. It provides a variety of optimizations, like overlapping chunks and compression. We see BlockJoin as complementary to the research in array databases and its ideas could be implemented to enhance their data loading and/or transformation.

Algebra Unifying Approaches. Kumar et al. [21] introduce learning generalized linear models over data residing in a relational database. The authors push parts of the computation of the ML model into joins over normalized data, similar to [12]. These works target generalized linear models only, while our approach subsumes a more generic optimization that can be used in arbitrary machine learning pipelines over normalized data. MLBase [20] provides high-level abstractions for ML tasks with basic support for relational operators. Their DSL allows the optimizer to choose different ML algorithm implementations, but does not take the relational operators into account nor does it optimize the physical representation of the data among different operators. Cohen et al. [14] execute linear algebra operations in a relational database, but do not present optimizations for block-partitioning the operands.

ML Libraries & Languages. SystemML's DML [7, 32, 8, 16], Mahout's Samsara [33], provide R-like linear algebra abstractions. SystemML executes locally or distributed on Hadoop and Spark, while Samsara targets Spark, Flink

and H_20. As there is no dedicated support for relational operators, ETL has to be executed using a different set of abstractions, and both systems lose potential for holistic optimization. MLlib [26, 10], MLI [34], Cumulon [18] and Pegasus [19] employ different strategies to efficiently execute matrix operations on distributed dataflow systems, but again do not target holistic optimization over relational and linear algebra operators. We presented recently the potential for optimizations across relational and linear algebra in the context of the Lara [22] language, based on Emma [4].

7. CONCLUSION & FUTURE WORK

In this paper, we introduce a scalable join algorithm for analytics which mix relational and linear algebra operations. Our technique reduces the re-partitioning overheads which stem from the different physical representations of relations and matrices. To this end, we propose BlockJoin, an optimized join algorithm, which fuses relational joins with blocked matrix partitioning, avoiding costly re-partitioning steps. We discuss different block materialization strategies of this join operator and their cost-model driven application, depending on the shape of the input data. In an extensive experimental evaluation, we show that BlockJoin outperforms the current state of the art implementation for dataflow systems up to a factor of six, and demonstrated that BlockJoin is scalable and robust on highly skewed data.

Future work. We plan to integrate BlockJoin and other physical operators into a common intermediate representation and optimizer which will be able to reason on mixed linear and relational algebra programs [4, 22]. Moreover, we plan to explore extensions of BlockJoin, to generate a variety of block-partitioned matrices for model selection workloads that are commonly employed to find well-working features and hyperparameters for machine learning models [32]. Furthermore, we plan future research to overcome the current limitation of BlockJoin to vectorization functions that can be executed separately on both relations.

Acknowledgments. This work has been supported through grants by the German Science Foundation (DFG) as FOR 1306 Stratosphere, by the German Ministry for Education and Research as Berlin Big Data Center BBDC (funding mark 01IS14013A), and by the European Union as Horizon 2020 projects Streamline (688191) and Proteus (687691).

8. REFERENCES

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *SIGMOD*, pages 967–980. ACM, 2008.

[2] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*. ACM, 2010.

[3] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10):1064–1075, 2012.

[4] A. Alexandrov et al. Implicit parallelism through deep language embedding. In *SIGMOD*, 2015.

[5] Apache Hadoop, http://hadoop.apache.org.

[6] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. In *Sigmod Record*, volume 27. ACM, 1998.

[7] M. Boehm et al. SystemML's optimizer: Plan generation for large-scale machine learning programs. *IEEE Data Eng. Bull.*, 37(3):52–62, 2014.

[8] M. Boehm et al. SystemML: Declarative machine learning on spark. *VLDB*, 9(13):1425–1436, 2016.

[9] P. A. Boncz, S. Manegold, M. L. Kersten, et al. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, 1999.

[10] R. Bosagh Zadeh et al. Matrix computations and optimization in apache spark. In *KDD*, pages 31–38. ACM, 2016.

[11] P. G. Brown. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*, pages 963–968. ACM, 2010.

[12] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *VLDB*, 1994.

[13] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *FMPC*, pages 120–127. IEEE, 1992.

[14] J. Cohen, , et al. Mad skills: new analysis practices for big data. *PVLDB*, 2(2):1481–1492, 2009.

[15] D. J. DeWitt et al. *Implementation techniques for main memory database systems*, volume 14. ACM, 1984.

[16] A. Elgohary et al. Compressed linear algebra for large-scale machine learning. *PVLDB*, 9(12):960–971, 2016.

[17] A. Ghoting et al. SystemML: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242. IEEE, 2011.

[18] B. Huang, S. Babu, and J. Yang. Cumulon: optimizing statistical data analysis in the cloud. In *SIGMOD*. ACM, 2013.

[19] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM*, 2009.

[20] T. Kraska et al. Mlbase: A distributed machine-learning system. In *CIDR*, volume 1, pages 2–1, 2013.

[21] A. Kumar, J. Naughton, and J. M. Patel. Learning generalized linear models over normalized data. In *SIGMOD*, pages 1969–1984. ACM, 2015.

[22] A. Kunft, A. Alexandrov, A. Katsifodimos, and V. Markl. Bridging the gap: Towards optimization across linear and relational algebra. BeyondMR, pages 1:1–1:4, 2016.

[23] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *ACM KDD*, 2005.

[24] Z. Li and K. A. Ross. Fast joins using join indices. *VLDB*, 8(1):1–24, 1999.

[25] R. Marek and E. Rahm. Tid hash joins. In *CIKM*, pages 42–49. ACM, 1994.

[26] X. Meng et al. Mllib: Machine learning in apache spark. *JMLR*, 17(34):1–7, 2016.

[27] J. K. Mullin. Optimal semijoins for distributed database systems. *IEEE Trans. Softw. Eng*, 16(5):558–560, 1990.

[28] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*. ACM, 2011.

[29] O. Polychroniou, R. Sen, and K. A. Ross. Track join: distributed joins with minimal network traffic. In *SIGMOD*, pages 1483–1494. ACM, 2014.

[30] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *ICDE*, pages 1194–1205. IEEE, 2016.

[31] N. Roussopoulos and H. Kang. A pipeline n-way join algorithm based on the 2-way semijoin program. *IEEE Trans. Knowl. Data Eng*, 3(4):486–495, 1991.

[32] S. Schelter et al. Efficient sample generation for scalable meta learning. In *ICDE*, 2015.

[33] S. Schelter et al. Samsara: Declarative machine learning on distributed dataflow systems. In *NIPS Workshop MLSystems*, 2016.

[34] E. R. Sparks et al. Mli: An api for distributed machine learning. In *ICDM*, pages 1187–1192. IEEE, 2013.

[35] J. W. Stamos and H. C. Young. A symmetric fragment and replicate algorithm for distributed joins. *IEEE Trans. Parallel Distrib. Syst*, 4(12):1345–1354, 1993.

[36] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *SIGMOD*. ACM, 2009.

[37] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *ACM SoCC*, 2011.

[38] M. Zaharia et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[39] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. *TODS*, 41(1):2, 2016.

[40] J. Zhou, P.-A. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In *ICDE*, pages 1060–1071. IEEE, 2010.