MNC: Structure-Exploiting Sparsity Estimation for Matrix Expressions

Johanna Sommer
IBM Germany

Matthias Boehm
Graz University of Technology

Alexandre V. Evfimievski
IBM Research – Almaden

Berthold Reinwald
IBM Research – Almaden

Peter J. Haas
UMass Amherst

ABSTRACT

Efficiently computing linear algebra expressions is central to machine learning (ML) systems. Most systems support sparse formats and operations because sparse matrices are ubiquitous and their dense representation can cause prohibitive overheads. Estimating the sparsity of intermediates, however, remains a key challenge when generating execution plans or performing sparse operations. These sparsity estimates are used for cost and memory estimates, format decisions, and result allocation. Existing estimators tend to focus on matrix products only, and struggle to attain good accuracy with low estimation overhead. However, a key observation is that real-world sparse matrices commonly exhibit structural properties such as a single non-zero per row, or columns with varying sparsity. In this paper, we introduce MNC (Matrix Non-zero Count), a remarkably simple, count-based matrix synopsis that exploits these structural properties for efficient, accurate, and general sparsity estimation. We describe estimators and sketch propagation for realistic linear algebra expressions. Our experiments—on a new estimation benchmark called SparsEst—show that the MNC estimator yields good accuracy with very low overhead. This behavior makes MNC practical and broadly applicable in ML systems.

ACM Reference Format:
Johanna Sommer, Matthias Boehm, Alexandre V. Evfimievski, Berthold Reinwald, and Peter J. Haas. 2019. MNC: Structure-Exploiting Sparsity Estimation for Matrix Expressions. In *2019 International Conference on Management of Data (SIGMOD '19), June 30–July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3299869.3319854

SIGMOD '19, June 30–July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3319854

1 INTRODUCTION

Modern machine learning (ML) systems aim to provide high-level languages with physical data independence, and automatically generate efficient execution plans to increase the productivity of data scientists and simplify deployment [10, 33, 65]. This separation of concerns hides the complexity of execution plans, local and distributed operations, as well as dense and sparse data layouts [21]. Language abstractions of these systems commonly rely on linear algebra such as matrix multiplications, reorganizations, aggregations, element-wise operations, and statistical functions, allowing users to create and customize ML algorithms and models.

Sources of Sparse Matrices: ML systems often support sparse matrix formats and operations because sparse matrices—with small ratio of non-zero values to matrix cells—are ubiquitous in many domains. Examples are natural language processing (NLP), graph analytics, recommender systems, and scientific computing, which often deal with ultra-sparse matrices of sparsity between 10^{-3} and 10^{-8}. Apart from sparse input data, there are other sources of sparse matrices. First, data pre-processing like one-hot encoding—which transforms a categorical feature of domain cardinality d into d 0/1 features—introduces correlated sparse columns [21]. Second, many element-wise operations can produce sparse intermediates even for dense inputs (e.g., selection predicates, or dropout layers). Third, transformation matrices like permutation and selection matrices—as used for random reshuffling and sampling via matrix products—are also huge, ultra-sparse matrices. Processing such ultra-sparse matrices using dense formats and operations would introduce prohibitively large storage and computation overheads. For this reason, open source systems like Julia [9], Spark MLlib [66], SystemML [10], and numerous prototypes [33, 65] automatically dispatch sparse operations when beneficial.

Problem of Sparsity Estimation: Estimating the sparsity of matrix expressions is an import yet challenging problem [43]. First, sparsity estimates are used during operation runtime for output format decisions and memory preallocation. Wrong decisions might largely impact memory requirements (e.g., wrong dense allocation of truly sparse outputs) and operation efficiency (e.g., wrong sparse allocation and

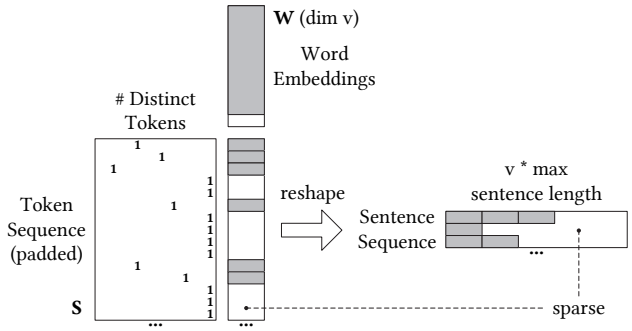


Figure 1: Example NLP Sentence Encoding.

updates of truly dense outputs). Second, sparsity estimates of matrix expressions are used during compilation for memory and cost estimates. Therefore, the accuracy of these estimates affect decisions on local versus distributed operations [10], matrix product chains [17], resource allocation [34], as well as rewrites and operator fusion [12, 20]. However, existing estimators have either prohibitive overheads or limited support for structural properties and thus, low accuracy.

Example Structural Properties: As an example of a sparse matrix product, Figure 1 shows an NLP scenario of encoding a sequence-of-words representation into word embeddings, as commonly used for deep learning. Input sentences are padded to the maximum sentence length and represented in a token sequence matrix $\mathbf{S}$. Ones in this matrix map a sequence position (i.e., row) to a distinct token (i.e., column). The last column of $\mathbf{S}$ represents unknowns such as pads and tokens that are not in the dictionary. Thus, $\mathbf{S}$ is typically very large and ultra sparse, due to skew of sentence lengths and large dictionaries. We multiply $\mathbf{S}$ with a pre-trained[1] word embeddings matrix $\mathbf{W}$ to yield an encoded token sequence. Finally, we reshape this matrix row-wise into the padded sentence representation which can be directly used for training or scoring of models like SentenceCNN [42]. The matrix product $\mathbf{S}\,\mathbf{W}$ has the special structural property that rows in $\mathbf{S}$ have exactly one non-zero. Similar structural properties are very common for selection matrices (e.g., for sampling) and permutation matrices (e.g., for random reshuffling). We shall show that these properties can be exploited to infer the *exact* output sparsity in this specific scenario, and generally improve the accuracy of sparsity estimation.

Contributions: Our primary contribution is a systematic exploitation of structural properties for better sparsity estimation of matrix products and other operations. To this end, we introduce the *Matrix Non-zero Count* (MNC) sketch and estimator, whose size is linear in the matrix dimensions. Compared to existing sparsity estimators [5, 10, 16, 39, 49, 65], we aim to achieve—as shown in Figure 2—high accuracy (with

[1]Word embeddings are commonly pre-trained—in an application-agnostic manner—with word2vec [25, 54, 55] over the Wikipedia corpus.

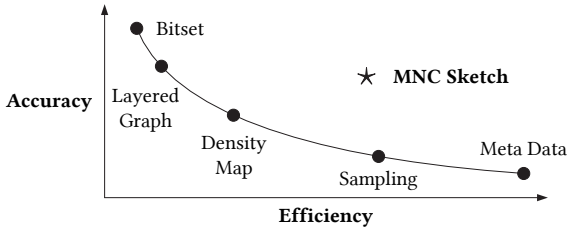


Figure 2: Accuracy/Efficiency Goal of the MNC Sketch.

exact results for special cases) with modest construction and estimation overhead. Our detailed contributions are:

- *Analysis of Sparsity Estimators:* We survey existing sparsity estimators in terms of their space and time complexity as well as potential bias in Section 2.
- *MNC Sketch:* We then introduce our novel MNC sketch and estimators for matrix products chains in Section 3. This also includes estimators for additional reorganization and element-wise operations in Section 4.
- *Sparsity Estimation Benchmark:* As a basis for our evaluation, we define a new benchmark for sparsity estimators—including a realistic mix of operations and matrices with structural properties—in Section 5.
- *Experiments:* Finally, in Section 6, we report on extensive experiments that compare MNC with existing estimators in terms of space efficiency, construction and estimation overhead, and estimation accuracy.

2 EXISTING SPARSITY ESTIMATORS

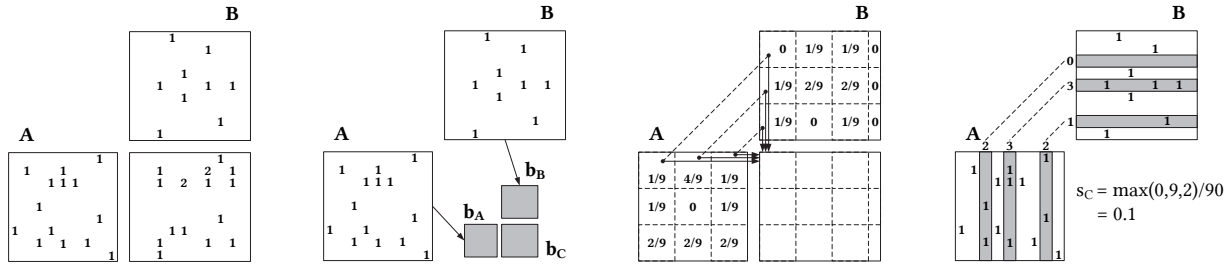
In this section, we briefly survey existing sparsity estimators for matrix products and product chains. Given an $m \times n$ matrix $\mathbf{A}$ with sparsity $s_A = \text{nnz}(\mathbf{A})/(mn)$—where $\text{nnz}(\mathbf{A})$ is the number of non-zeros in $\mathbf{A}$—and an $n \times l$ matrix $\mathbf{B}$ with sparsity s_B, we aim to estimate the sparsity $\hat{s}_C$ of the matrix product $\mathbf{C} = \mathbf{A}\,\mathbf{B}$. Further, let $d = \max(m, n, l)$ be the maximum dimension. Similarly, for a chain of matrix products $\mathbf{M}^{(1)}\,\mathbf{M}^{(2)} \ldots \mathbf{M}^{(k)}$, we are interested in sparsity estimates s_{ij} for subchains $\mathbf{M}^{(i)} \ldots \mathbf{M}^{(j)}$ with $1 \le i < j \le k$.

Assumptions: All existing estimators as well as our MNC estimator make—implicitly or explicitly—the following two simplifying but reasonable assumptions:

- *A1: No Cancellation Errors:* Positive and negative values could cancel each other to zero on aggregation. Similarly, multiplying very small values may create zeros due to round-off errors.
- *A2: No Not-A-Number (NaN) Values:* NaNs are challenging for sparsity estimation and sparse linear algebra in general because $\text{NaN} \cdot 0 = \text{NaN}$ [43].

2.1 Naïve Estimators

The naïve estimators are extremes in a spectrum of potential estimators with different runtime and accuracy tradeoffs.



(a) Matrix Product $\mathbf{C} = \mathbf{A}\,\mathbf{B}$ (b) Bitset Estimator (c) Density Map Estimator (d) Sampling-based Estimator

Figure 3: Examples of Existing Sparsity Estimators.

Naïve Metadata Estimators: The naïve metadata estimators (e.g., in SystemML) derive the output sparsity solely from the sparsity of the input matrices [10, 16, 39, 65]. These estimators are best-effort choices because the metadata is readily available during compilation without additional runtime overhead. In detail, we distinguish two types. First, the unbiased, average-case estimator $\mathbf{E}_{\text{ac}}$ [10] is

$$\hat{s}_C = 1 - (1 - s_A \cdot s_B)^n, \tag{1}$$

which assumes uniformly distributed non-zeros, and estimates the output sparsity as the complementary probability of an output cell being zero[2]. In contrast, the worst-case estimator $\mathbf{E}_{\text{wc}}$ [10]—that aims to provide an upper bound for worst-case memory estimates—is

$$\begin{aligned}\hat{s}_C &= \min(1, \text{nnz}(\mathbf{A})/m) \cdot \min(1, \text{nnz}(\mathbf{B})/l) \\ &= \min(1, s_A \cdot n) \cdot \min(1, s_B \cdot n),\end{aligned} \tag{2}$$

which assumes an adversarial pattern where $\mathbf{A}$ and $\mathbf{B}$ have aligned column and row vectors of non-zeros, respectively.

Naïve Bitset Estimator: The other extreme is the naïve *bitset* estimator $\mathbf{E}_{\text{bmm}}$ (e.g., in NVIDIA cuSPARSE and Intel MKL [49], and similarly in SciDB [61]) that constructs—as shown in Figure 3(b)—boolean matrices $\mathbf{b_A}$ and $\mathbf{b_B}$ and performs an exact boolean matrix multiply $\mathbf{b_C} = \mathbf{b_A}\,\mathbf{b_B}$. Such a product can be efficiently computed via bitwise AND (multiply) and OR (sum). The exact sparsity estimate is

$$s_C = \hat{s}_C = \text{bitcount}(\mathbf{b_C})/(ml), \tag{3}$$

where $\text{bitcount}(\mathbf{b_C})$ is the number of set bits in the output. Although the size is 64x smaller than in dense double precision, this estimator adds significant size and runtime overhead.

2.2 Density Map Estimator

To account for sparsity skew with configurable runtime overhead, the *density map* estimator $\mathbf{E}_{\text{dm}}$ creates density maps $\mathbf{dm_A}$ and $\mathbf{dm_B}$ for the input matrices $\mathbf{A}$ and $\mathbf{B}$ [39]. Such a density map imposes a block size b (by default, $b = 256$) and stores the sparsity of individual $b \times b$ blocks. The density map of the output $\mathbf{dm_C}$ is computed via a pseudo-matrix-multiplication over density maps $\mathbf{dm_A}\,\mathbf{dm_B}$ as follows:

$$\begin{aligned}\mathbf{dm}_{C_{ij}} &= \oplus_{k=1}^{n/b}\, \mathbf{E}_{\text{ac}}(\mathbf{dm}_{A_{ik}}, \mathbf{dm}_{B_{kj}}) \\ &\text{with } s_{A \oplus B} = s_A + s_B - s_A s_B,\end{aligned} \tag{4}$$

which replaces multiply with the average-case estimator $\mathbf{E}_{\text{ac}}$ and plus with a formula for probabilistic propagation. Finally, we obtain the output sparsity $\hat{s}_C$ from the total number of non-zeros in $\mathbf{dm_C}$ (i.e., scaled density map by block cells). For example, assuming the example matrices from Figure 3(a), we construct the density maps shown in Figure 3(c) and compute the output density map accordingly.

Effect of Block Size: Intuitively, the block size b allows a trade-off between accuracy and runtime overhead [39]. Moreover, the density map is a generalization of the naïve estimators because for $b = 1$ it is equivalent to the bitset estimator ($\mathbf{E}_{\text{dm}} \equiv \mathbf{E}_{\text{bmm}}$), while for $b = d$ it is equivalent to the average-case estimator ($\mathbf{E}_{\text{dm}} \equiv \mathbf{E}_{\text{ac}}$). However, during our experiments we made the interesting observation that—for special structural properties—smaller block sizes can lead to higher errors. For example, consider a 200×100 matrix $\mathbf{A}$ with 50 non-zeros arranged as a column vector ($s_A = 0.0025$) and a dense 100×100 matrix $\mathbf{B}$ ($s_B = 1$). The true number of non-zeros is 5,000 but with block sizes $b = 200$, $b = 100$, and $b = 50$, we estimate 4,429, 3,942, and 3,179. In this scenario, there are no collisions but the smaller the block size, the higher the estimated probability of collisions.

Dynamic Block Sizes: The fixed block size is problematic for ultra-sparse matrices because a moderate default can render the density map larger than the input. A natural extension would be dynamic density maps that adapt local block sizes to the non-zero structure, for example, via a recursive quad tree partitioning as done in the AT-Matrix [40] for NUMA-aware partitioning. However, the non-aligned blocks in $\mathbf{dm_A}$ and $\mathbf{dm_B}$ would complicate the estimator.

2.3 Sampling-based Estimators

The sampling-based estimator $\mathbf{E}_{\text{smpl}}$ also aims to account for sparsity skew [65]. In contrast to the density map, this

[2] An even simpler estimator for ultra-sparse inputs is $\hat{s}_C = s_A s_B n$ [16].

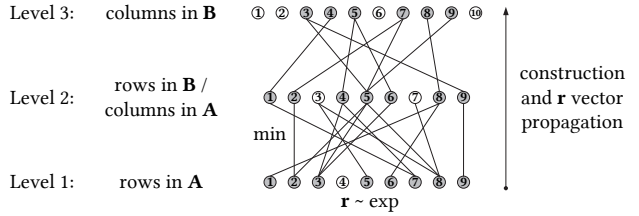


Figure 4: Example Existing Graph-based Estimator.

estimator draws a random and uniformly distributed sample $\mathcal{S}$ of columns from $\mathbf{A}$ and aligned rows from $\mathbf{B}$. For example, Figure 3(d) shows a sample of three columns and rows from $\mathbf{A}$ and $\mathbf{B}$. The output sparsity is then estimated as

$$\hat{s}_C = \max_{k \in \mathcal{S}} \left(\text{nnz}(\mathbf{A}_{:k}) \cdot \text{nnz}(\mathbf{B}_{k:})\right)/(ml). \quad (5)$$

Intuitively, this estimator views a matrix product as the sum of outer products $\mathbf{A}_{:k}\,\mathbf{B}_{k:}$ and uses the sparsity of the largest outer product as its estimate. Hence, $\mathbf{E}_{\text{smpl}}$ is biased in terms of being a strict lower bound of the true output sparsity, which has the undesirable property of not converging to the true sparsity s_C, even if $|\mathcal{S}| = n$. Appendix A describes an inexpensive extension to overcome these issues, as well as a hybrid estimator [5] that relies on hashing and sampling.

2.4 Graph-based Estimators

In a seminal paper on estimating the non-zero structure of matrix product chains, Cohen introduced a so-called *layered graph* to estimate the number of non-zeros per rows and columns in the output [16]. This algorithm is a specialized variant of estimating the size of the transitive closure [15].

Layered Graph: The layered graph $\mathcal{G}$ of a chain of matrices $(\mathbf{M}_1, \mathbf{M}_2, \ldots, \mathbf{M}_k)$ has $k+1$ levels. Nodes in level 1 refer to rows in $\mathbf{M}_1$, level 2 to columns in $\mathbf{M}_1$ and rows in $\mathbf{M}_2$, level 3 to columns in $\mathbf{M}_2$ and rows in $\mathbf{M}_3$, etc. Edges represent the position of non-zeros in $\mathbf{M}_i$ by connecting their specific row to column nodes. For example, Figure 4 shows the layered graph for our product $\mathbf{C} = \mathbf{A}\,\mathbf{B}$ from Figure 3(a), where unreachable nodes (shown in white) are discarded.

Sparsity Estimate: The graph-based estimator $\mathbf{E}_{\text{gph}}$ then assigns $\mathbf{r}$-vectors—of configurable size—drawn from the exponential distribution with $\lambda = 1$ to all leaf nodes in level 1. The $\mathbf{r}$-vectors are propagated bottom-up through $\mathcal{G}$ by computing $\mathbf{r}$ of an inner node as the element-wise minimum of the $\mathbf{r}$-vectors of its inputs. The final estimate is then

$$\hat{s}_C = \left(\sum_{v \in \text{ roots}} \frac{|\mathbf{r}_v| - 1}{\text{sum}(\mathbf{r}_v)} \right) / (ml), \quad (6)$$

which estimates the non-zeros per output column, as well as aggregates and scales them to the overall output sparsity. Note that for matrix product chain optimization, the estimated column and row counts can be directly used to compute sparsity-aware costs of subchains [16].

Table 1: Analysis of Existing Sparsity Estimators.

Estimator	Space	Time	🔗	Bias
MetaAC $\mathbf{E}_{\text{ac}}$	$O(1)$	$O(1)$	✓	
MetaWC $\mathbf{E}_{\text{wc}}$	$O(1)$	$O(1)$	✓	$\overline{s_C}$
Bitset $\mathbf{E}_{\text{bmm}}$	$O(mn + nl + ml)$	$O(mnl)$	✓	
DMap $\mathbf{E}_{\text{dm}}$	$O(\frac{mn+nl+ml}{b^2})$	$O(\frac{mnl}{b^3})$	✓	
Sample $\mathbf{E}_{\text{smpl}}$	$O(\lvert\mathcal{S}\rvert)$	$O(\lvert\mathcal{S}\rvert(m+l))$		$\underline{s_C}$
LGraph $\mathbf{E}_{\text{gph}}$	$O(rd + \text{nnz}(\mathbf{A},\mathbf{B}))$	$O(r(d + \text{nnz}(\mathbf{A},\mathbf{B})))$	✓	
MNC $\mathbf{E}_{\text{mnc}}$	$O(d)$	$O(d + \text{nnz}(\mathbf{A},\mathbf{B}))$	✓	

2.5 Discussion and Analysis

Finally, we summarize the existing estimators, and discuss the relationship to join cardinality estimators.

Analysis of Sparsity Estimators: Table 1 summarizes the surveyed sparsity estimators, which all—except the sampling-based estimator—apply to both matrix products and matrix product chains (🔗). Overall, these estimators cover a spectrum that trades runtime efficiency and estimation accuracy. First, the constant-time average- and worst-case estimators $\mathbf{E}_{\text{ac}}$ and $\mathbf{E}_{\text{wc}}$ are extremely cheap to compute if overall sparsity is readily available. However, $\mathbf{E}_{\text{ac}}$ assumes uniformity and independence, which rarely hold in practice, while $\mathbf{E}_{\text{wc}}$ uses a conservative estimate that is biased to over-estimation (upper bound). Second, the bitset and density map estimators $\mathbf{E}_{\text{bmm}}$ and $\mathbf{E}_{\text{dm}}$ require space and time proportional to dense input sizes. While $\mathbf{E}_{\text{bmm}}$ provides a constant reduction of 64 (compared to double precision), $\mathbf{E}_{\text{dm}}$ reduces space and time requirements by b^2 and b^3 with configurable block size b. However, both can quickly exceed the space requirements of ultra-sparse inputs. Third, the sampling-based estimator is relatively inexpensive but only applies to single matrix products, can cause repeated matrix multiplication for lazily evaluated distributed operations, and is biased to under-estimation (lower bound). Fourth, the graph-based estimator $\mathbf{E}_{\text{gph}}$ is able to estimate the non-zero structure of matrix product chains and thus, provides very good accuracy. However, the graph construction and usage is linear in the number of non-zeros (times the size of $\mathbf{r}$-vectors) which can cause significant runtime overhead, compared to well-optimized matrix multiplication kernels.

Join Cardinality Estimation: Assuming matrices $\mathbf{A}_{ikv_1}$ and $\mathbf{B}_{kjv_2}$ are given as 3-column relations of row indexes, column indexes, and values, sparsity estimation of the product $\mathbf{AB}$ is equivalent to estimating the cardinality of the join-group-by query $\gamma_{ij}(\mathbf{A} \bowtie_k \mathbf{B})$. Hence, existing cardinality estimators directly apply. For example, the recently proposed index-based join sampling [45] is very similar to the sampling-based estimator (see Section 2.3) that systematically samples related rows from $\mathbf{A}$ and columns from $\mathbf{B}$. However, existing cardinality estimators largely ignore—similar to existing sparsity estimators—structural properties.

Error Propagation: Given the close relationship to cardinality estimation, a natural question is if errors propagate similarly. Ioannidis and Christodoulakis established that cardinality estimation errors propagate exponentially through joins [35]. We use a similar maximum error argument. Assume a matrix product chain $(\mathbf{M}_1, \mathbf{M}_2, \ldots, \mathbf{M}_k)$ of $n \times n$ matrices and constant sparsity s. Considering uniformly distributed non-zeros, and $1-(1-s_{\mathbf{M}_{12}} \cdot s_{\mathbf{M}_3})^n$ from Equation (1) together with a constant error of ϵ, we substitute $s_{\mathbf{M}_{12}}$ with $(1+\epsilon) \cdot s_{\mathbf{M}_{12}}$. Then, we see that ϵ propagates exponentially in the dimension n and in the number of products $k-1$. Despite this exponential propagation, sparsity estimation is feasible in practice because matrix expressions often exhibit structural properties that can be exploited.

3 MNC SKETCH

We now introduce the *matrix non-zero count* (MNC) sketch by describing the main data structure, as well as estimators for matrix products and product chains. Motivated by our experience with sparse matrices in practice, and inspired by several aspects of Cohen's estimator [16], we base our MNC framework on count-based histograms of the number of non-zeros (NNZ) per row and column, as well as additional metadata to encode common structural properties.

3.1 MNC Framework

Data Structure: The MNC sketch $\mathbf{h_A}$ of an $m \times n$ matrix $\mathbf{A}$ comprises the following information, where we use $\mathbf{h}$ as a shorthand whenever the context is clear.

- *Row/Column NNZs:* Count vectors $\mathbf{h}^r = \text{rowSums}(\mathbf{A} \neq 0)$ and $\mathbf{h}^c = \text{colSums}(\mathbf{A} \neq 0)$ indicate the NNZs per row and column, where $\mathbf{h}^r_i$ is the count of the ith row.
- *Extended Row/Column NNZs:* Count vectors $\mathbf{h}^{er} = \text{rowSums}((\mathbf{A} \neq 0) \cdot (\mathbf{h}^c = 1))$ and $\mathbf{h}^{ec} = \text{colSums}((\mathbf{A} \neq 0) \cdot (\mathbf{h}^r = 1))$ indicate the NNZs per row/column that appear in columns/rows with a single non-zero.
- *Summary Statistics:* Metadata includes the maximum NNZ per row $\max(\mathbf{h}^r)$ and column $\max(\mathbf{h}^c)$, the number of non-empty rows $\text{nnz}(\mathbf{h}^r)$ and columns $\text{nnz}(\mathbf{h}^c)$, the number of half-full rows $|\mathbf{h}^r > n/2|$ and columns $|\mathbf{h}^c > n/2|$, as well as flags for diagonal matrices.

Construction and Analysis: The construction of the count vectors $\mathbf{h}^r$ and $\mathbf{h}^c$ is done in a single scan over the non-zeros of $\mathbf{A}$, where we aggregate the respective row and column counts. Sparse representations such as the row-major CSR (compressed sparse rows) readily provide $\mathbf{h}^r$ as metadata, which makes the construction slightly more efficient. We then materialize the summary statistics in a single pass over $\mathbf{h}^r$ and $\mathbf{h}^c$. Finally, if $\max(\mathbf{h}^r) > 1$ or $\max(\mathbf{h}^c) > 1$, we construct the extended count vectors $\mathbf{h}^{er}$ and $\mathbf{h}^{ec}$ in a second scan over the non-zeros of $\mathbf{A}$ with filter conditions

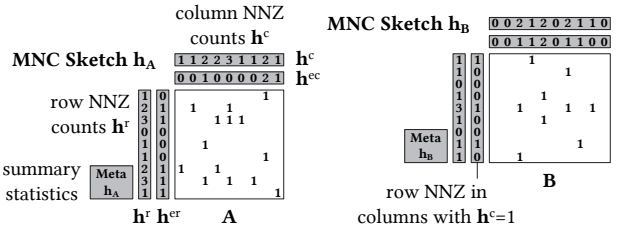


Figure 5: Example MNC Sketches.

$\mathbf{h}^c = 1$ and $\mathbf{h}^r = 1$, respectively. Similar to histogram construction during data ingestion [2], the MNC construction can be piggybacked on the read of matrices, which often uses two passes as well (e.g., multi-threaded CSV read, or sparse binary read). To summarize, the MNC sketch construction requires $O(\text{nnz}(\mathbf{A}) + m + n)$ time[3] (linear in the non-zeros and dimensions of $\mathbf{A}$), while the size of an MNC sketch is $O(m+n)$ (linear in the dimensions of $\mathbf{A}$). The small size of $\mathbf{h_A}$ also makes it amenable to large-scale ML, where the sketch can be computed via distributed operations and subsequently, collected and used in the driver for compilation.

Examples: Figure 5 shows the sketches $\mathbf{h_A}$ and $\mathbf{h_B}$ for our running example matrix product $\mathbf{C} = \mathbf{A}\,\mathbf{B}$. We observe that the non-zero structure is captured in a fairly fine-grained manner while bounding the size overhead to the size of the dimensions. Similarly, the MNC sketch for our introductory example from Figure 1, would capture the structural property $\max(\mathbf{h}^r) = 1$ and individual row and column NNZ counts, which are important for accurate sparsity estimation.

3.2 Sparsity Estimation

Having defined the structure of MNC sketches, we now discuss how these sketches are used for sparsity estimation. We first present individual components and their underlying foundations before synthesizing the final sparsity estimator.

Basic Sparsity Estimation: In general, an MNC sketch $\mathbf{h_A}$ can be viewed as a special density map with overlapping blocks covering rows and columns in $\mathbf{A}$ but by encoding non-zero counts of rows and columns separately, it is better suited to handle sparsity skew across columns [21]. Accordingly, we can directly apply $\mathbf{E}_{\text{dm}}$ as a fallback estimator over $\mathbf{h}^c_\mathbf{A}$ and $\mathbf{h}^r_\mathbf{B}$ (i.e., with $m \times l$ output block size)[4]. Additionally, however, we exploit structural properties as follows.

Theorem 3.1. *Given MNC sketches $\mathbf{h_A}$ and $\mathbf{h_B}$ for matrices $\mathbf{A}$ and $\mathbf{B}$, the output sparsity $s_\mathbf{C}$ of the matrix product $\mathbf{C} = \mathbf{A}\,\mathbf{B}$ can be exactly computed under the assumptions A1 and A2 via a dot product of $\mathbf{h}^c_\mathbf{A}$ and $\mathbf{h}^r_\mathbf{B}$:*

$$s_\mathbf{C} \equiv \hat{s}_\mathbf{C} = \mathbf{h}^c_\mathbf{A}\,\mathbf{h}^r_\mathbf{B}/(ml) \text{ if } \max(\mathbf{h}^r_\mathbf{A}) \leq 1 \vee \max(\mathbf{h}^c_\mathbf{B}) \leq 1. \quad (7)$$

[3] Dense formats require a scan over all mn cells but these formats are only used if $s_A \geq 0.4$ (i.e., $\text{nnz}(\mathbf{A}) \geq 0.4mn$).

[4] Alternatively, we could also use $\mathbf{E}_{\text{dm}}$ over $\mathbf{h}^r_\mathbf{A}$ and $\mathbf{h}^c_\mathbf{B}$ (with 1×1 output block size) but this would increase the estimation costs from $O(n)$ to $O(ml)$.

Proof. Under the assumptions A1 and A2, the sparsity s_C of $\mathbf{C} = \mathbf{A}\,\mathbf{B}$ is equal to the sparsity of a boolean matrix product $(\mathbf{A} \neq 0)\,(\mathbf{B} \neq 0)$. Such boolean products can be represented as $Z = \bigcup_j \mathcal{A}_j \times \mathcal{B}_j$, where $\mathcal{A}_j$ are row indexes of column $(\mathbf{A}_{:j} \neq 0)$ and $\mathcal{B}_j$ are column indexes of row $(\mathbf{B}_{j:} \neq 0)$ [5]. For $\max(\mathbf{h}_\mathbf{A}^r) \leq 1$, each row index appears in at most one $\mathcal{A}_j$. Similarly, for $\max(\mathbf{h}_\mathbf{B}^c) \leq 1$, each column index appears in at most one $\mathcal{B}_j$. Hence, the union Z is distinct. The size of $\mathcal{A}_j \times \mathcal{B}_j$ is $\mathbf{h}_{\mathbf{A}j}^c \cdot \mathbf{h}_{\mathbf{B}j}^r$. Since the union is distinct, we have $|Z| = \sum_j |\mathcal{A}_j \times \mathcal{B}_j|$. Combining these facts, we get $s_C = |Z|/(ml) = (\sum_j \mathbf{h}_{\mathbf{A}j}^c \cdot \mathbf{h}_{\mathbf{B}j}^r)/(ml)$, which is exactly $\mathbf{h}_\mathbf{A}^c\,\mathbf{h}_\mathbf{B}^r/(ml)$. □

Exploiting Extended NNZ Counts: Theorem 3.1 only applies if all rows or columns contain at most one non-zero. In practice, we also encounter matrices with a subset[5] of qualifying rows or columns. We aim to exploit these properties by fine-grained extended row and column NNZ counts:

$$\hat{s}_C = (\mathbf{h}_\mathbf{A}^{ec}\,\mathbf{h}_\mathbf{B}^r + (\mathbf{h}_\mathbf{A}^c - \mathbf{h}_\mathbf{A}^{ec})\,\mathbf{h}_\mathbf{B}^{er} + \mathbf{E}_{\text{dm}}(\mathbf{h}_\mathbf{A}^c - \mathbf{h}_\mathbf{A}^{ec}, \mathbf{h}_\mathbf{B}^r - \mathbf{h}_\mathbf{B}^{er}, p) \cdot p)/(ml), \tag{8}$$

where the first term $\mathbf{h}_\mathbf{A}^{ec}\,\mathbf{h}_\mathbf{B}^r + (\mathbf{h}_\mathbf{A}^c - \mathbf{h}_\mathbf{A}^{ec})\,\mathbf{h}_\mathbf{B}^{er}$ represents the non-zeros that are exactly known, while the second term is a generic estimator of the remaining quantities. There are no side effects between these terms because both sets of non-zeros are disjoint. For this reason, we use for the remaining quantities—instead of ml—a reduced output size p:

$$p = (m - |\mathbf{h}_\mathbf{A}^r = 1|) \cdot (l - |\mathbf{h}_\mathbf{B}^c = 1|), \tag{9}$$

where $|\mathbf{h}_\mathbf{A}^r = 1|$ is the number of rows with one non-zero.

Additional Lower and Upper Bounds: In addition to the aforementioned basic and extended estimators, we establish lower and upper bounds that guard against low accuracy for matrix products with adversarial non-zero structure. Here, $\text{nnz}(\mathbf{h}_\mathbf{A}^r)$ denotes the number of non-empty rows in $\mathbf{A}$.

Theorem 3.2. *Given MNC sketches $\mathbf{h}_\mathbf{A}$ and $\mathbf{h}_\mathbf{B}$ for matrices $\mathbf{A}$ and $\mathbf{B}$, the output sparsity s_C of the matrix product $\mathbf{C} = \mathbf{A}\,\mathbf{B}$ is bounded under the assumptions A1 and A2 by:*

$$s_C \geq |\mathbf{h}_\mathbf{A}^r > n/2| \cdot |\mathbf{h}_\mathbf{B}^c > n/2|/(ml)$$
$$s_C \leq nnz(\mathbf{h}_\mathbf{A}^r) \cdot nnz(\mathbf{h}_\mathbf{B}^c)/(ml). \tag{10}$$

Proof. Under assumptions A1 and A2, we have again a boolean matrix product $Z = \bigcup_j \mathcal{A}_j \times \mathcal{B}_j$. First, the quantities $\text{nnz}(\mathbf{h}_\mathbf{A}^r)$ and $\text{nnz}(\mathbf{h}_\mathbf{B}^c)$ determine the number of distinct row indexes $\bigcup_j \mathcal{A}_j$ and distinct column indexes $\bigcup_j \mathcal{B}_j$. Hence, $s_C = |Z|/(ml)$ is upper bounded by $\text{nnz}(\mathbf{h}_\mathbf{A}^r) \cdot \text{nnz}(\mathbf{h}_\mathbf{B}^c)/(ml)$. Second, any pair of rows in $\mathbf{A}_{i:}$ and columns in $\mathbf{B}_{:j}$ with $\text{nnz}(\mathbf{A}_{i:}) > n/2 \wedge \text{nnz}(\mathbf{B}_{:j}) > n/2$ is guaranteed to yield a non-zero cell $\mathbf{C}_{ij}$ because at least one element-wise multiplication $\mathbf{A}_{ik} \cdot \mathbf{B}_{kj}$ with $1 \leq k \leq n$ will be $(\mathbf{A}_{ik} \neq 0) \cdot (\mathbf{B}_{kj} \neq 0)$. Hence, s_C is lower bounded by $|\mathbf{h}_\mathbf{A}^r > n/2| \cdot |\mathbf{h}_\mathbf{B}^c > n/2|/(ml)$. □

[5]For example, as of 2019/04/04, 25,800,660 (89%) of the 28,929,182 public GitHub repositories have 0 or 1 stars (estimates obtained via `is:public`).

Algorithm 1 MNC Sparsity Estimation

Input: MNC sketches $\mathbf{h}_\mathbf{A}$ and $\mathbf{h}_\mathbf{B}$ for matrices $\mathbf{A}$ and $\mathbf{B}$
Output: Output sparsity s_C

1: *// a) basic and extended sparsity estimation, incl upper bound*
2: **if** $\max(\mathbf{h}_\mathbf{A}^r) \leq 1 \vee \max(\mathbf{h}_\mathbf{B}^c) \leq 1$ **then** *// see Theorem 3.1*
3: $\quad \text{nnz} \leftarrow \mathbf{h}_\mathbf{A}^c\,\mathbf{h}_\mathbf{B}^r$
4: **else if** $\text{exists}(\mathbf{h}_\mathbf{A}^{er}) \vee \text{exists}(\mathbf{h}_\mathbf{B}^{er})$ **then** *// extended NNZ counts*
5: $\quad \text{nnz} \leftarrow \mathbf{h}_\mathbf{A}^{ec}\,\mathbf{h}_\mathbf{B}^r + (\mathbf{h}_\mathbf{A}^c - \mathbf{h}_\mathbf{A}^{ec})\,\mathbf{h}_\mathbf{B}^{er}$ *// exact fraction*
6: $\quad p \leftarrow (\text{nnz}(\mathbf{h}_\mathbf{A}^r) - |\mathbf{h}_\mathbf{A}^r = 1|) \cdot (\text{nnz}(\mathbf{h}_\mathbf{B}^c) - |\mathbf{h}_\mathbf{B}^c = 1|)$ *// #cells*
7: $\quad \text{nnz} \leftarrow \text{nnz} + \mathbf{E}_{\text{dm}}(\mathbf{h}_\mathbf{A}^c - \mathbf{h}_\mathbf{A}^{ec}, \mathbf{h}_\mathbf{B}^r - \mathbf{h}_\mathbf{B}^{er}, p) \cdot p$ *// generic rest*
8: **else** *// generic fallback estimate*
9: $\quad p \leftarrow \text{nnz}(\mathbf{h}_\mathbf{A}^r) \cdot \text{nnz}(\mathbf{h}_\mathbf{B}^c)$ *// #cells*
10: $\quad \text{nnz} \leftarrow \mathbf{E}_{\text{dm}}(\mathbf{h}_\mathbf{A}^c, \mathbf{h}_\mathbf{B}^r, p) \cdot p$
11: *// b) apply lower bound, see Theorem 3.2*
12: $\text{nnz} \leftarrow \max(\text{nnz}, |\mathbf{h}_\mathbf{A}^r > n/2| \cdot |\mathbf{h}_\mathbf{B}^c > n/2|)$ *// lower bound*
13: **return** $s_C \leftarrow \text{nnz}/(ml)$

Sparsity Estimator: Putting it altogether, Algorithm 1 shows our MNC sparsity estimator $\mathbf{E}_{\text{mnc}}$, which requires $O(n)$ time (linear in the common dimension of $\mathbf{A}\,\mathbf{B}$). First, in lines 1-10, we apply the basic and extended estimators. The first two cases exploit coarse- or fine-grained structural properties, but if unavailable, we fall back to a density-map-like estimator over column and row counts. Entries of non-existing extended count vectors are treated as zeros. The upper bound is imposed via a modified output size $p = \text{nnz}(\mathbf{h}_\mathbf{A}^r) \cdot \text{nnz}(\mathbf{h}_\mathbf{B}^c)$, which also improves the estimation of collisions. Second, in lines 11-13, we impose the lower bound and return s_C as the estimated NNZ scaled by the output size ml.

3.3 Sketch Propagation

So far we only discussed sparsity estimation for a single matrix product. However, for chains of matrix products, we need to derive sketches for intermediates as well, to allow for recursive sparsity estimation of arbitrary subchains.

Basic Sketch Propagation: Figure 6 gives an overview of our sketch propagation—which requires $O(d)$ time—for deriving a sketch $\mathbf{h}_\mathbf{C}$ from input sketches $\mathbf{h}_\mathbf{A}$ and $\mathbf{h}_\mathbf{B}$. First, we compute the output sparsity $\hat{s}_C$ as described in Section 3.2. Second, we derive row and column histograms from $\mathbf{A}$ and $\mathbf{B}$ by scaling them with the relative change of NNZ from $\mathbf{A}$ to $\mathbf{C}$ and from $\mathbf{B}$ to $\mathbf{C}$ as shown in Equation(11). This ensures consistency regarding $\sum \mathbf{h}_\mathbf{C}^r \approx \hat{s}_C ml$ and $\sum \mathbf{h}_\mathbf{C}^c \approx$

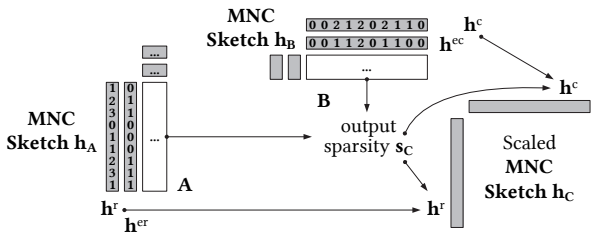


Figure 6: Basic Sketch Propagation.

$\hat{s}_C ml$ but assumes that the distribution of non-zeros per row and column propagate over the matrix product.

$$\begin{aligned} \mathbf{h}^r_C &= \text{round}(\mathbf{h}^r_A \cdot \hat{s}_C ml / \sum \mathbf{h}^r_A), \\ \mathbf{h}^c_C &= \text{round}(\mathbf{h}^c_B \cdot \hat{s}_C ml / \sum \mathbf{h}^c_B). \end{aligned} \tag{11}$$

Probabilistic Rounding: For ultra-sparse matrices, basic rounding can introduce significant bias. For example, consider a special case where every entry in $\mathbf{h}^r_C$ (without rounding) is 0.4. With rounding, we get 0.0 non-zeros for all rows, which leads to the wrong assumption of an empty intermediate and thus, empty final output. Hence, we use probabilistic rounding with probabilities $\mathbf{h}^r_C - \lfloor \mathbf{h}^r_C \rfloor$ of rounding entries in $\mathbf{h}^r_C$ up. This approach ensures unbiased propagation with minimal variance, which worked well in practice.

Exact Sketch Propagation: Additionally, we exploit the metadata for exact sketch propagation. If either **A** or **B** is fully diagonal (all non-zeros and no zero on diagonal) and square, we propagate the full sketch of the other input because the output characteristics are guaranteed identical:

$$\mathbf{h}_C = \begin{cases} \mathbf{h}_A & \text{if } \mathbf{B} \text{ is diagonal} \\ \mathbf{h}_B & \text{if } \mathbf{A} \text{ is diagonal.} \end{cases} \tag{12}$$

Implementation Details: With these techniques, we are able to compute sketches and the output sparsity of DAGs of matrix products. These DAGs are given in an intermediate representation (IR) where nodes are input matrices (leafs) or operations, edges are data dependencies, and we are interested in estimating the sparsity of—potentially multiple—root nodes of the DAG. The MNC sketches of leaf nodes are built from the input matrices, which can be done as part of reoptimization similar to MatFast [65] or SystemML [10]. Then, we compute and propagate sketches bottom-up through the DAG, with three simple yet effective improvements. First, we memoize—i.e., materialize and reuse—intermediate sketches because nodes might be reachable over multiple paths. Similarly, in the context of matrix multiplication chain optimization, we reuse intermediate sketches across overlapping sub-problems (see Appendix C). Second, we do not propagate sketches to the root nodes but directly estimate their sparsity. Third, we use special cases for matrix self-products to avoid unnecessary sketch construction.

4 ADDITIONAL OPERATIONS

Chains of pure matrix products rarely exceed a length of five in real numerical computing and ML workloads. Much more common are chains of matrix products interleaved with reorganization operations such as transpose or reshape, and element-wise matrix operations. In this section, we, therefore, extend the MNC Sketch and related sketch propagation techniques for these additional operations. Except for naïve metadata estimators in Matlab [24] and SystemML [10], this problem has not been addressed in the literature.

Scope of Supported Operations: In addition to matrix products, we support reorganization operations that change the position of values, and element-wise operations. First, reorganizations include transpose (matrix transposition), reshape (row-wise change of dimension sizes), diag (vector placement/extraction onto/from the diagonal), and rbind/cbind (row-/column-wise concatenation of matrices). Furthermore, this includes the comparisons $\mathbf{A} = 0$ and $\mathbf{A} \neq 0$ that extract the zero or non-zero structure. Second, we also support element-wise addition $\mathbf{A}+\mathbf{B}$ and multiplication $\mathbf{A} \odot \mathbf{B}$.

4.1 Sparsity Estimation

Similar to Section 3, we first investigate the simpler problem of estimating sparsity of a single operation. To simplify notation, we use $s(x, m, n) = x/(mn)$ to compute the sparsity from non-zeros x and dimensions $m \times n$.

Reorganization Operations: Sparsity estimation for reorganizations is straightforward because they allow—with few exceptions—for exact inference of the number of non-zeros and thus, sparsity from metadata. For transpose, reshape, and $\mathbf{A} \neq 0$, we have $s_C = s_A$, while for diag (vector-matrix) we have $s_C = s(\text{nnz}(\mathbf{A}), m, m)$. Similarly, the sparsity s_C for rbind, cbind, and $\mathbf{A} = 0$ is exactly computed via $s(\text{nnz}(\mathbf{A})+\text{nnz}(\mathbf{B}), m_A+m_B, n_A)$, $s(\text{nnz}(\mathbf{A})+\text{nnz}(\mathbf{B}), m_A, n_A+n_B)$, and $1 - s_A$, respectively.

Element-wise Operations: Element-wise addition and multiplication are more challenging. A baseline solution would pick the row or column dimension, and apply per slice the average case estimates $s_{Ai} + s_{Bi} - s_{Ai} \cdot s_{Bi}$ and $s_{Ai} \cdot s_{Bi}$ for addition and multiplication. However, this approach would not fully exploit the structural information of the entire MNC sketch. Instead, we use both row and column count vectors $\mathbf{h}^r$ and $\mathbf{h}^c$ of both **A** and **B** as follows:

$$\begin{aligned} s_C = \frac{1}{m \cdot n} \begin{cases} \sum_{i=0}^{m} \left(\mathbf{h}^r_{Ai} + \mathbf{h}^r_{Bi} - \mathbf{h}^r_{Ai} \cdot \mathbf{h}^r_{Bi} \cdot \lambda \right) & + \\ \sum_{i=0}^{m} \left(\mathbf{h}^r_{Ai} \cdot \mathbf{h}^r_{Bi} \cdot \lambda \right) & \odot, \end{cases} \\ \text{where } \lambda = \sum_{j=0}^{n} \left(\mathbf{h}^c_{Aj} \cdot \mathbf{h}^c_{Bj} \right) / (\text{nnz}(\mathbf{A}) \cdot \text{nnz}(\mathbf{B})). \end{aligned} \tag{13}$$

Intuitively, we aggregate row-wise estimates of non-zeros but scale them by λ, which represents collisions of non-zeros, computed from column counts. This computation is algebraically symmetric, so aggregating column-wise estimates and scaling by row collisions yields the exact same result.

4.2 Sketch Propagation

For estimating sparsity of entire chains of operations, we need to propagate sketches over these operations too. Sketch

propagation is more involved as we need to preserve individual row and columns counts. In this context, we always propagate count vectors, but we propagate extension vectors only when they are known to be exactly preserved.

Reorganization Operations: Several reorganizations still allow retaining valuable structural properties when propagating MNC sketches. First, for $\mathbf{C} = \mathbf{A} \neq 0$, we simply propagate $\mathbf{h}_\mathbf{C} = \mathbf{h}_\mathbf{A}$. Second, transpose, rbind (and symmetrically cbind), diag (vector-to-matrix-diag), and $\mathbf{A} = 0$ can also be exactly derived with the following formulas:

$$\begin{aligned}
\text{transpose: } & \mathbf{h}^r_\mathbf{C} = \mathbf{h}^c_\mathbf{A}, \mathbf{h}^{er}_\mathbf{C} = \mathbf{h}^{ec}_\mathbf{A}, \mathbf{h}^c_\mathbf{C} = \mathbf{h}^r_\mathbf{A}, \mathbf{h}^{ec}_\mathbf{C} = \mathbf{h}^{er}_\mathbf{A} \\
\text{rbind: } & \mathbf{h}^r_\mathbf{C} = \text{rbind}(\mathbf{h}^r_\mathbf{A}, \mathbf{h}^r_\mathbf{B}), \mathbf{h}^{er}_\mathbf{C} = \emptyset \\
& \mathbf{h}^c_\mathbf{C} = \mathbf{h}^c_\mathbf{A} + \mathbf{h}^c_\mathbf{B}, \mathbf{h}^{ec}_\mathbf{C} = \mathbf{h}^{ec}_\mathbf{A} + \mathbf{h}^{ec}_\mathbf{B} \\
\text{diag: } & \mathbf{h}^r_\mathbf{C} = \mathbf{h}^{er}_\mathbf{C} = \mathbf{h}^c_\mathbf{C} = \mathbf{h}^{ec}_\mathbf{C} = \mathbf{h}^r_\mathbf{A} \\
\mathbf{A} = 0\text{: } & \mathbf{h}^r_\mathbf{C} = n - \mathbf{h}^r_\mathbf{A}, \mathbf{h}^c_\mathbf{C} = m - \mathbf{h}^c_\mathbf{A}, \mathbf{h}^{er}_\mathbf{C} = \mathbf{h}^{ec}_\mathbf{C} = \emptyset
\end{aligned} \tag{14}$$

Third, reshaping an $m \times n$ matrix $\mathbf{A}$ into a $k \times l$ matrix $\mathbf{C}$ requires a more fine-grained approach. For the sake of a simple presentation, we focus on row-wise reshaping where $m \mod k = 0$, that is, where multiple input rows are concatenated to a single output row. In a first step, we compute $\mathbf{h}^r_\mathbf{C}$ by aggregating every m/k row counts, which—at conceptual level—is equivalent to $\mathbf{h}^r_\mathbf{C} = \text{rowSums}(\text{matrix}(\mathbf{h}^r_\mathbf{A}, k, m/k))$. In a second step, we then scale and replicate column counts to compute $\mathbf{h}^c_\mathbf{C}$, which is similar to $\mathbf{h}^c_\mathbf{C} = \text{rep}(\text{round}(\mathbf{h}^c_\mathbf{A}/(m/k)), m/k)$. Finally, remaining operations such as matrix-to-vector diag are handled in a best-effort manner, which is sufficient for practical purposes as the output is a vector.

Element-wise Operations: Propagating MNC sketches for element-wise addition and multiplication follows the same approach as described for Equation (13) in Section 4.1, but now we materialize individual row/column estimates. To enable this symmetric computation, we first prepare both scaling factors λ^r and λ^c and then compute the output MNC sketch—but only for count vectors—as follows:

$$\begin{aligned}
+\text{: } & \mathbf{h}^r_\mathbf{C} = \text{round}(\mathbf{h}^r_\mathbf{A} + \mathbf{h}^r_\mathbf{B} - \mathbf{h}^r_\mathbf{A} \cdot \mathbf{h}^r_\mathbf{B} \cdot \lambda^c), \\
& \mathbf{h}^c_\mathbf{C} = \text{round}(\mathbf{h}^c_\mathbf{A} + \mathbf{h}^c_\mathbf{B} - \mathbf{h}^c_\mathbf{A} \cdot \mathbf{h}^c_\mathbf{B} \cdot \lambda^r) \\
\odot\text{: } & \mathbf{h}^r_\mathbf{C} = \text{round}(\mathbf{h}^r_\mathbf{A} \cdot \mathbf{h}^r_\mathbf{B} \cdot \lambda^c), \mathbf{h}^c_\mathbf{C} = \text{round}(\mathbf{h}^c_\mathbf{A} \cdot \mathbf{h}^c_\mathbf{B} \cdot \lambda^r)
\end{aligned} \tag{15}$$

Similar to sketch propagation for matrix products, we apply probabilistic rounding to guard against systematic bias.

Implementation Details: Efficient sketch propagation further draws from a careful implementation. First, we use shallow copies of sketches and internal arrays whenever sketch components directly propagate to avoid unnecessary copies and garbage collection overhead. Examples are transpose, diag, $\mathbf{X} \neq 0$, and special cases of matrix multiplications. Second, we also derive the summary metadata, and only fallback to recomputation over $\mathbf{h}^r_\mathbf{C}$ and $\mathbf{h}^c_\mathbf{C}$ if this is not possible.

5 SPARSITY ESTIMATION BENCHMARK

As a basis for a systematic evaluation, we define the SparsEst benchmark for sparsity estimation of matrix operations and matrix expressions that covers existing and new use cases. In the interest of reproducibility, we base this benchmark solely on synthetic and publicly available real datasets.

Benchmark Metrics: Existing sparsity estimators trade off accuracy and runtime as visualized in Figure 2. Hence, we made the design choice of including both aspects separately:

- *M1 Accuracy:* A common accuracy metric is the *absolute ratio error* (ARE) defined as $|\hat{s}_\mathbf{C} - s_\mathbf{C}|/s_\mathbf{C}$ [8, 26, 29]. This metric is asymmetric as it penalizes overestimation more than under-estimation. Hence, we use the *relative error*[6] defined as $\max(\hat{s}_\mathbf{C}, s_\mathbf{C})/\min(\hat{s}_\mathbf{C}, s_\mathbf{C})$ [14]. For multiple experiments, we additively aggregate $\hat{S}_\mathbf{C} = \sum_i^n \hat{s}_{\mathbf{C}i}$ (or equivalently non-zeros) and compute the final error as $\max(\hat{S}_\mathbf{C}, s_\mathbf{C} n)/\min(\hat{S}_\mathbf{C}, s_\mathbf{C} n)$.
- *M2 Estimation Time:* The runtime metric is the total estimation time covering both sketch construction and estimation, which can be reported separately.

An additional useful but optional metric is the *M3 Total Runtime*—including sketch construction, estimation, and plan execution—because sparsity estimation influences plan costs and the sketches could be exploited during runtime. However, the exploitation of matrix sketches during optimization and runtime is beyond the scope of this paper.

Benchmark Use Cases: In detail, our benchmark suite consists of the following three major groups of use cases.

- *B1 Structured Matrix Products (Struct):* Synthetic matrix products with specific structural properties.
- *B2 Real Matrix Operations (Real):* Real sparse matrix operations from NLP, graphs, and pre-processing.
- *B3 Real Matrix Expressions (Chain):* Pure and mixed chains of matrix products and other operations.

The categories B1 and B2 apply to all sparsity estimators, no matter if matrix product chains or other operations are supported. Chaining these operations further allows for a simple evaluation of how errors propagate.

B1 Structured Matrix Products (Struct): The category Struct aims to test—using synthetically generated data (Syn)—specific structural properties that commonly occur in practice or constitute challenging special cases. First, B1.1 represents the NLP sentence encoding scenario from Figure 1, where $\mathbf{W}$ is dense except an empty last row and $\mathbf{X}$ is a 0/1 matrix whose NNZ per column are generated from a power law distribution, except the last column that contains a fraction α non-zeros. Thus, the output sparsity is $(1 - \alpha)$ independent of the dimensions of $\mathbf{X}$ and $\mathbf{W}$. Second, B1.2 and B1.3 emulate the scaling and random reshuffling of a matrix,

[6] A similar normalization for the ARE is $|\hat{s}_\mathbf{C} - s_\mathbf{C}|/\min(\hat{s}_\mathbf{C}, s_\mathbf{C})$.

Table 2: Overview of Benchmark Use Cases.

Structured Matrix Products (Struct)			Real Matrix Products (Real)			Real Matrix Product Chains (Chain)		
Name	**Expression**	**Data**	**Name**	**Expression**	**Data**	**Name**	**Expression**	**Data**
B1.1 NLP	$\mathbf{X}\,\mathbf{W}$	Syn1	B2.1 NLP	$\mathbf{X}\,\mathbf{W}$	AMin A	B3.1 NLP	reshape($\mathbf{X}\,\mathbf{W}$)	AMin A
B1.2 Scale	diag(λ) $\mathbf{X}$	Syn2	B2.2 Project	$\mathbf{X}\,\mathbf{P}$	Cov	B3.2 S&S	$\mathbf{S}^\top\,\mathbf{X}^\top$ diag($\mathbf{w}$) $\mathbf{X}\,\mathbf{S}\,\mathbf{B}$	Mnist1m
B1.3 Perm	table($\mathbf{s1}$, $\mathbf{s2}$) $\mathbf{X}$	Syn2	B2.3 CoRefG	$\mathbf{G}\,\mathbf{G}^\top$	AMin R	B3.3 Graph	$\mathbf{P}\,\mathbf{G}\,\mathbf{G}\,\mathbf{G}\,\mathbf{G}$	AMin R
B1.4 Outer	$\mathbf{C}\,\mathbf{R}$	Syn3/Syn4	B2.4 EmailG	$\mathbf{G}\,\mathbf{G}$	Email	B3.4 Rec	$(\mathbf{P}\,\mathbf{X} \neq 0) \odot (\mathbf{P}\,\mathbf{L}\,\mathbf{R}^\top)$	Amazon
B1.5 Inner	$\mathbf{R}\,\mathbf{C}$	Syn4/Syn3	B2.5 Mask	$\mathbf{M} \odot \mathbf{X}$	Mnist1m	B3.5 Pred	$\mathbf{X} \odot ((\mathbf{R} \odot \mathbf{S} + \mathbf{T}) \neq 0)$	Mnist1m

which are key primitives. Since the left-hand-side matrices are a fully diagonal and a random permutation matrix, the output sparsity is equivalent to the input $\mathbf{X}$. Third, B1.4 and B1.5 represent special cases [39] with square matrices $\mathbf{C}$ and $\mathbf{R}$ that contain a single dense column and aligned row (and vice versa), and thus, result in a fully-dense or almost empty (single non-zero) matrix, respectively.

B2 Real Matrix Operations (Real): Furthermore, the second category Real investigates real matrix products and element-wise operations. First, B2.1 encodes—similar to B1.1—the AMin A dataset of paper abstracts. Second, B2.2 projects columns of the Covertype dataset that is known to have columns with varying sparsity [21]. This column projection is a common operation in feature selection workloads. Third, B2.3 performs co-reference counting via a matrix product of AMin R with its transposed representation. Fourth, B2.4 is a simple self matrix product for email network analysis. Fifth and finally, B2.5 performs image masking on the Mnist dataset via an element-wise multiplication.

B3 Real Matrix Expressions (Chain:) The category B3 further represents several interesting, real matrix expressions that are composed of matrix products and other operations. First, B3.1 extends our NLP example from B1.1 and B2.1 by matrix reshaping from token-embeddings to sentence-embeddings as used in our introductory example in Figure 1. Second, B3.2 shows a matrix product chain for scaling and shifting of $\mathbf{X}$ as used in the inner loops of regression and classification algorithms to avoid densifying sparse matrices via shifting upfront. Here, $\mathbf{S}$ is a special scale and shift matrix of dimensions $n \times n$ with dense diagonal and last row, where n is the number of features of $\mathbf{X}$. Third, B3.3 investigates—similar to existing benchmarks [16, 39, 57]—a chain of matrix self-products (also known as matrix powers) as used for reachability queries, and other graph analytics [38]. The matrix $\mathbf{P}$ is a constructed selection matrix. Fourth, B3.4 shows the computation of recommendations for known ratings of selected users $\mathbf{P}$ via a learned low rank factorization $\mathbf{L}$ and $\mathbf{R}$. Fifth, B3.5 uses only element-wise operations to apply a complex predicate for image masking. This combined mask selects all $\mathbf{X}_{ij}$ where $(\mathbf{R}_{ij} \wedge \mathbf{S}_{ij}) \vee \mathbf{T}_{ij}$ evaluates to true. This is a common pattern found in spatial data processing via linear algebra, where $\odot$ replaces $\wedge$, $+$ (or max) replaces $\vee$, and $\neq 0$ ensures a 0/1 indicator matrix for value extraction.

Table 3: Overview of Used Datasets.

Name	Rows m	Columns n	Nnz $\|\mathbf{X}\|_0$	**Sparsity** s_X
Amazon [52]	8M	2.3M	22.4M	0.0000012
AMin A [4]	25.1M	2.5M	25.1M	0.00000039
AMin R[4]	3.1M	3.1M	25.2M	0.0000026
Cov [48]	581K	54	6.9M	0.22
Email [46]	265K	265K	420K	0.000006
Mnist1m [13]	1M	784	202M	0.25

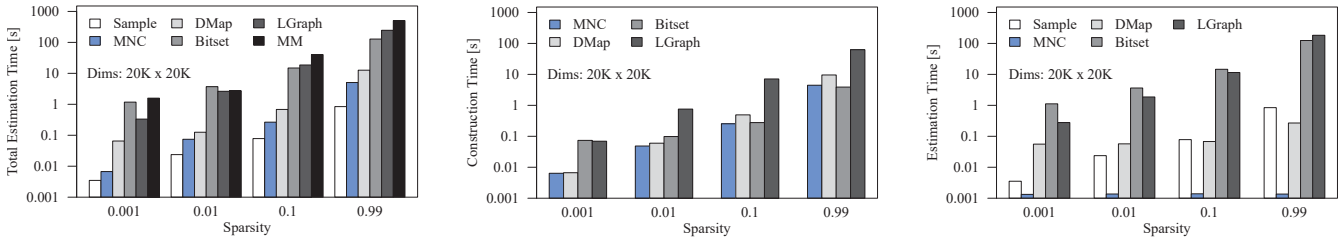
Datasets: Table 3 summarizes the used datasets, which come from a mix of different domains and exhibit diverse data characteristics. First, Amazon [30, 52], Cov [48], and Mnist1m [13, 50], are the well-known Amazon books review datasets, UCI Covertype, and the Mnist dataset of hand-written digits with 1M rows. Second, AMin [4, 60, 62] refers to the AMiner dataset, specifically the DBLP-Citation-network (V10, October 2017). From this publication dataset, we created (1) AMin R, the paper reference matrix, and (2) AMin A, a token-sequence matrix of paper abstracts. For AMin A, we extracted all 17,646,972 sentences with a total of 374,677,894 tokens, padded the sentences to the maximum length of 2,508 tokens, and encoded the tokens as column positions regarding a Wikipedia embeddings dictionary of size 2,518,950. For handling purposes, we selected a subset of 10K sentences. Third, Email [46, 47] is an Email network from an EU research institution, covering 18 months and 3,038,531 emails between 287,755 (1,258 local) email addresses.

6 EXPERIMENTS

Our experiments study the MNC sketch in comparison to existing sparsity estimators with regard to (1) construction and estimation overheads, (2) accuracy for matrix products, and (3) accuracy for chains of matrix products and other operations. In this context, we aim to explore a variety of data characteristics, parameter configurations, and operation workloads utilizing the introduced SparsEst benchmark.

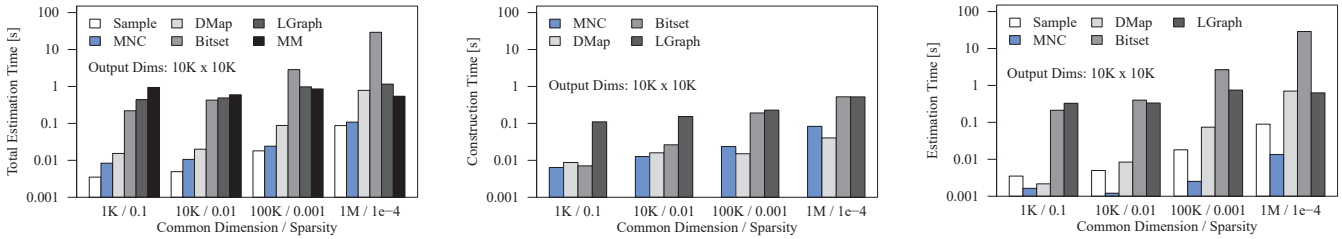
6.1 Experimental Setup

HW and SW Setting: We ran our experiments on a 2+10 node cluster of two head nodes, and ten worker nodes. All nodes have two Intel Xeon E5-2620 CPUs @ 2.10 GHz-2.50 GHz (24 virtual cores), 128 GB DDR3 RAM, a nominal peak memory bandwidth of 2×43 GB/s, and run CentOS



(a) Total Estimation Time (M2) (b) Construction Time (c) Estimation Time

Figure 7: Construction and Estimation Runtime for Varying Sparsity.



(a) Total Estimation Time (M2) (b) Construction Time (c) Estimation Time

Figure 8: Construction and Estimation Runtime for Varying Dimensions.

Linux 7.2. We used OpenJDK 1.8.0_151, and HDP 2.5 with Apache Hadoop 2.7.3. All experiments ran as single-node YARN applications with 80 GB max and initial JVM heap sizes, which allowed us to parallelize up to ten experiments. Unless otherwise stated, we report the mean of 20 repetitions.

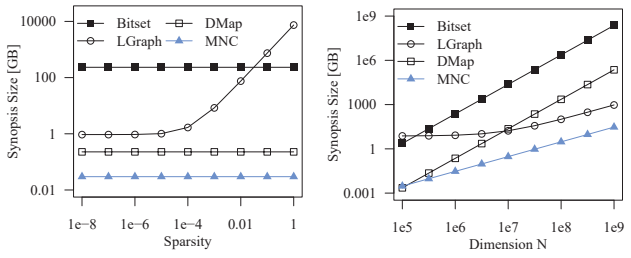
Baselines: For the sake of a fair comparison, we implemented all sparsity estimators[7] from Section 2 in the same framework, and extended these estimators—if necessary and feasible—for all operations and sketch propagation. To obtain the ground truth output sparsity, we execute FP64 matrix operations with internal dispatch of dense and sparse operations. These operations also serve as an additional baseline in terms of efficiency, because any sketch construction and estimation overhead should be well below the operation runtime. We consistently use multi-threaded FP64 matrix multiplications (MM), but single-threaded sketch construction and estimation. While our MNC sketch is trivial to parallelize over row or column partitions, multi-threading would put more complex baselines such as the layered graph at a disadvantage. Internally, the density map is implemented via our default FP64 matrix, while the bitset estimator, uses either a linearized long array—to facilitate cache blocking—or Java's `BitSet` implementation per row, as well as bitwise OR's of 64 values at a time in the inner loop. Finally, for the density map, layered graph, and sampling-based estimators, we use by default a block size $b = 256$, an $\mathbf{r}$-vector length 32, and a sample fraction $f = 0.05$ unless otherwise stated.

[7] MNC and the baselines are open source at `github.com/apache/systemml`. We also considered specialized baselines for encoding word embeddings, but primitives in TensorFlow [1] and PyTorch [58] return dense tensors.

6.2 Construction and Estimation

In a first series of experiments, we investigate the construction and estimation overhead in terms of runtime and synopsis size of the various sparsity estimators. Recall from Figure 2, our MNC sketch is designed to reach efficiency close to sampling, but significantly better than the bitset, density map, and layered graph. We measure the total runtime of sparsity estimation, which assumes in-memory input matrices with basic metadata, and includes input sketch construction if necessary. Note that we exclude the metadata estimators $\mathbf{E}_{\text{wc}}$ and $\mathbf{E}_{\text{ac}}$ because they only require few scalar floating-point operations, independent of the data size.

Runtime with Varying Sparsity: Figure 7 shows the estimation runtime for a product of two random matrices with fixed dimension of $20\text{K} \times 20\text{K}$ and varying sparsity in $[10^{-3}, 0.99]$ and thus, increasing number of non-zeros. We avoid a sparsity of 1.0 because special cases for fully dense matrix products would apply. Figure 7(a) shows the total estimation time—including construction and estimation—for all estimators. On one side of the spectrum, the metadata estimators are of course by far the fastest estimators and hence, excluded from the plot. MNC then comes close to sampling and generally outperforms the density map. On the other side of the spectrum, we have the bitset and layered graph, where the layered graph has advantages with low sparsity because it requires time proportional to the number of non-zeros. Despite single-threaded construction and estimation, all estimators rarely exceed the runtime of multi-threaded matrix multiplications. Figures 7(b) and 7(c) separately show the construction and estimation overheads, where $\mathbf{E}_{\text{smpl}}$ requires



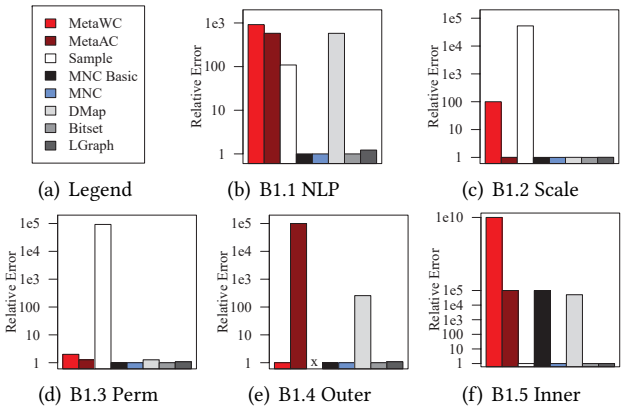
(a) Varying Sparsity (b) Varying Dimensions

Figure 9: Analytical Synopses Size Overhead.

no construction because the sample is not materialized. The bitset and density map spent—proportionally to their total runtime—less time in construction because they are dominated by matrix-product-like estimation with cubic complexity. All estimators exploit—independent of their sketch representation—sparsity to prune unnecessary operations.

Runtime with Varying Dimensions: In contrast to the previous experiment, Figure 8 shows the runtime with fixed number of non-zeros (1M) and output dimensions (10K × 10K) but varying common dimension and sparsity. Apart from similar trends, there are three observations. First, with increasing sparsity, the bitset and density map become less competitive even compared to a full matrix multiplication. Second, both sampling and MNC show similar scaling as their estimation time depends on the common dimension. Third, MNC's construction time scales slightly worse than the density map in this scenario with rectangular dimensions and high sparsity because its size reduction per row (10K) is smaller than the density map's $256^2 = 64$K reduction.

Size Overhead: To better understand the estimation and construction runtimes, Figure 9 shows analytical results for the size overhead of estimators with synopses. Asymptotically, the bitset and density map are dense sketches of size proportional to the number of matrix cells, whereas MNC and the layered graph are of size $O(d)$ (in the dimensions) and $O(d + \text{nnz}(\mathbf{A}, \mathbf{B}))$ (in the dimensions and non-zeros), respectively. However, we aim to show the absolute factors that matter in practice. First, Figure 9(a) shows the results for constant dimensions $m = n = 1$M and varying sparsity in $[10^{-8}, 1]$. MNC requires $2 \cdot 4 \cdot 1\text{M} \cdot 4\,\text{B} = 32\,\text{MB}$, while bitset and density map require 125 GB and 122 MB. The large difference between bitset and density map originate from the $64x$ versus 256^2x size reduction. The layered graph has a more complex behavior. For small sparsity, it is dominated by the size of nodes (in the size of dimensions) with each node having an **r**-vector of size 32. As the sparsity increases, however, the size gets dominated by the edges (in the number of non-zeros) even exceeding the size requirements of the bitset. Second, Figure 9(b) shows the results for a constant number of non-zeros (1G) and increasing dimensions and thus,



(a) Legend (b) B1.1 NLP (c) B1.2 Scale

(d) B1.3 Perm (e) B1.4 Outer (f) B1.5 Inner

Figure 10: Accuracy Results for B1 Struc.

decreasing sparsity. Here, we see the expected asymptotic behavior with several interesting break-even points. However, only the MNC sketch ensures very low size overhead—note that a single y-axis tic represents three orders of magnitude—across the entire spectrum of scenarios.

6.3 Accuracy for Matrix Products

In a second set of experiments, we investigate the estimation accuracy for matrix products on SparsEst B1 and B2. We report the relative error, which is bounded by $[1, \infty)$. Appendix A includes further experiments that compare MNC with additional sampling-based estimators.

Accuracy B1 Struct: Figure 10 shows the relative error of all baselines for B1. We also include MNC Basic, which is the MNC estimator without extension vectors and bounds. In detail, we configured these benchmarks as follows:

- *B1.1 NLP:* A 100K × 100K token matrix, a fraction of 0.001 known tokens, 300-dimensional embeddings.
- *B1.2 Scale:* A diagonal 100K×100K matrix and 100K×2K matrix with sparsity 0.01.
- *B1.3 Perm:* A random 100K × 100K permutation matrix and 100K × 2K matrix with sparsity 0.5.
- *B1.4/B1.5:* Matrices **R** and **C** of size 100K × 100K.

For these structured matrix inputs, we observe that the metadata estimators, sampling, and density map generally show large errors. The $\mathbf{E}_{\text{wc}}$ estimator typically performs—due to its conservative approach—worse than $\mathbf{E}_{\text{ac}}$. B1.4 is the only exception where $\mathbf{E}_{\text{wc}}$ outperforms $\mathbf{E}_{\text{ac}}$ because two ultra-sparse matrices produce a fully dense output. Furthermore, the sampling-based estimator $\mathbf{E}_{\text{smpl}}$ suffers from its bias, which—except for B1.5—does not hold. On B1.4 $\mathbf{E}_{\text{smpl}}$ computed $10^{10}/0 = \infty$ because with sample fraction of 0.05 only 2 out of 20 repetitions sampled the dense **C** and **R** vectors. While the density map performs good on uniformly distributed sparsity (e.g., B1.2 and B1.3), it shows large errors for structured matrices like B1.1, B1.4, and B1.5 because the

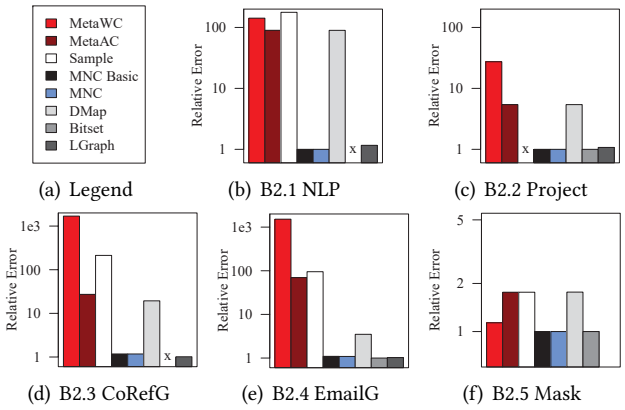


Figure 11: Accuracy Results for B2 Real.

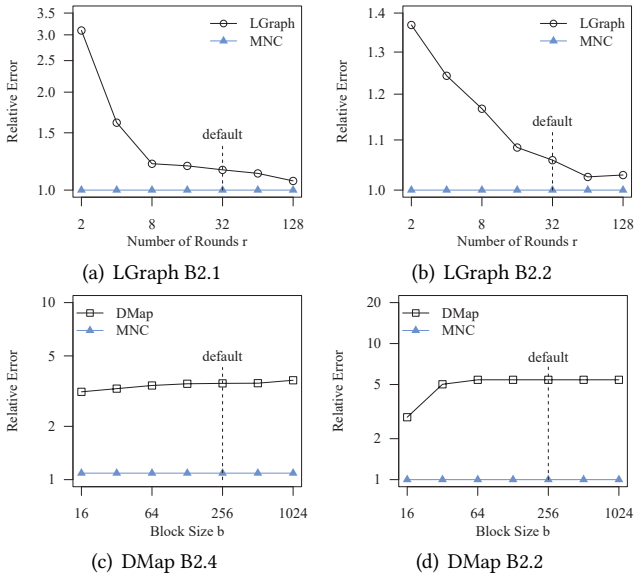


Figure 12: Accuracy with Varying Parameters.

square blocks are not able to preserve row and column-wise structure. In contrast, the layered graph is very accurate with a maximum observed error of 1.61 (average 1.22) for B1.1. However, only bitset and MNC yielded exact results for all B1 scenarios; B1.5 is an example where MNC relied on its upper bound $\text{nnz}(\mathbf{h}_\mathbf{A}^r) \cdot \text{nnz}(\mathbf{h}_\mathbf{B}^c)$ to accomplish that.

Accuracy B2 Real: In addition, Figure 11 shows the accuracy results of all baselines for the matrix products B2.1-B2.4. We used the real input datasets as described in Section 5, and the projection matrix **P** for B2.2 extracts columns of the range [11, 50], which are all dummy coded and thus, ultra-sparse. The metadata estimators, sampling, and the density map show again relatively large errors because they struggle to encode the structure of these matrices. For example, the density map fails to recognize the varying sparsity in B2.2 because the block size of 256 × 256 is too coarse-grained for the Cov dataset with 54 columns. In contrast, MNC computes the exact sparsity for B2.1 and B2.2, and shows even for co-reference counting and email graph analysis small errors of 1.17 and 1.09, respectively. The layered graph also consistently yields low errors, and outperforms MNC on co-reference counting. Although the bitset estimator always computes the exact sparsity, it fails for B2.1 and B2.3 because its size exceeds the available memory. For example, creating a bitset for AMin A for B2.1 would require ≈ 8 TB.

MNC Extension Vectors: While the impact of MNC extension vectors was negligible for B1 and B2, we observed improvements of up to 48.1% on other datasets.

6.4 Accuracy for Other Operations

The final scenario of B2 Real is a pure element-wise multiplication B2.5 for image masking, which does not apply to the layered graph so we exclude this estimator. Figure 11(f) shows the accuracy of the remaining estimators. We used Mnist1m and applied a mask **M** that selects the 14×14 center of all 28×28 images. This can be seen as an adversarial mask, because most non-zeros appear in the center region. Similar to most other image datasets, Mnist represents images as rows in matrix **X**. Thus, pixel masking is a column-wise operation. Exploiting this structure, MNC again yields the exact result. In contrast, both the density map and sampling have a relative error of ≈ 1.76. Given the adversarial pattern, MetaWC—with its worst-case guarantee—also performs very well with a relative error of 1.13.

6.5 Baseline Parameter Configurations

One strength of our MNC sketch is that it does not require parameters to trade-off between estimation overhead and accuracy. However, to better understand the default configurations of the baseline estimators, we conduct a systematic evaluation of the layered graph and density map.

Layered Graph: The layered graph uses **r**-vectors to hold random numbers—drawn from an exponential distribution—for each node. Increasing the vector length (the number of "rounds"), decreases the relative error in expectation, but also linearly increases the estimation runtime. Figures 12(a) and 12(b) show the resulting relative error with varying number of rounds for B2.1 and B2.2 for which MNC yields exact results. We see knees that are data-dependent but the default of 32 usually attains very good accuracy. Note that on datasets like AMin A—with significant structure—the relative error differences can be quite large.

Density Map: Similarly, the density map's block size b determines the granularity of the retained sparsity structure because sparsity is maintained for $b \times b$ blocks. Increasing the block size leads to more aggregation and thus, likely lower accuracy but also significantly decreased overhead due to the

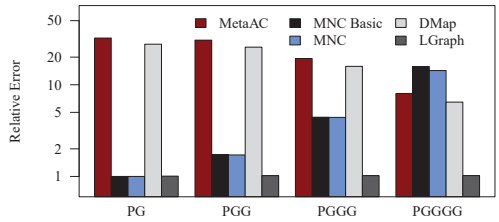


Figure 13: Accuracy Results for B3.3 Graph.

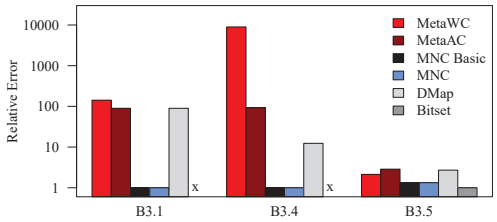


Figure 14: Accuracy Results for B3 Chain.

quadratic influence. Figures 12(c) and 12(d) show the relative error with increasing block size for B2.4 and B2.2. Here, B2.4 replaces B2.1 because, with small block sizes, the density map ran out of memory for the large B2.1 NLP matrix AMin A. Overall, we see a rather small influence on the resulting error. Note that for B2.2, only block sizes 16 and 32 can exploit the column structure for Cov with 54 columns. Since, the density map requires matching block sizes of all inputs for matrix products and other operations, this data-dependent configuration becomes a real challenge for arbitrary DAGs of matrix operations with unknown sparsity of intermediates.

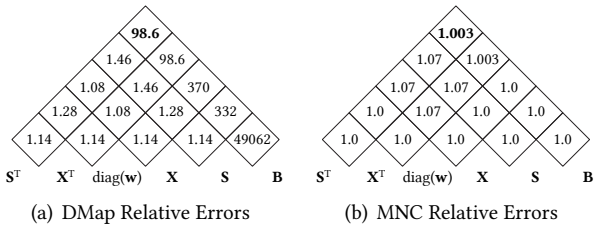
6.6 Accuracy for Chains of Operations

In a last set of experiments, we study the accuracy of MNC and other estimators for entire matrix expressions using the benchmark category B3 Chain. We start by showing a negative result for matrix powers and then discuss the results of more realistic matrix expressions.

Matrix Powers (B3.3): The benchmark query B3.3 consists of a pure chain of matrix products $\mathbf{P}\,\mathbf{G}\,\mathbf{G}\,\mathbf{G}\,\mathbf{G}$ including the fourth matrix power. Matrix powers is a common benchmark [16, 39, 57]—although closed-form solutions via inverse exist [38]—and thus, it applies to all existing estimators, except the bitset because a boolean matrix of $\mathbf{G}$ exceeds the available memory. In detail, we constructed a selection matrix $\mathbf{P}$ to extract the top 200 papers by number of references from $\mathbf{G}$. Then, we perform three more products with $\mathbf{G}$ to determine transitively referenced papers over three hops. Figure 13 shows the accuracy results for all intermediates and the final output. There are four interesting observations. First, the layered graph yields very good accuracy with only slightly increasing errors, but at the cost of impractical overhead as shown in Figures 7 and 8. Second, MNC computes the exact sparsity for the initial selection, whereas the MetaAC and density map estimators assume uniformity and thus, fail to capture the structure of extracted rows. Third, maybe surprising for a reader, MetaAC and density map show *decreasing* errors with increasing chain length. The reason is that matrix powers are densifying operations that systematically increase the number of non-zeros and uniformity. In contrast, our MNC estimator shows increasing errors for longer chains and thus, is outperformed by both MetaAC and density map. Our sketch propagation aims to propagate structure as much as possible, which is counter-productive in scenarios like this with vanishing structure. Fourth, MNC outperforms MNC basic (14.3 vs. 15.8) because the upper bound helps determining the feasible output size p.

Mixed Expressions: The benchmark queries B3.1, B3.4, and B3.5 include a mix of matrix products, element-wise operations, and reorganizations. Hence, these benchmarks do not apply to the layered graph. Furthermore, the bitset estimator failed to represent the ultra-sparse matrices in B3.1 and B3.4, which would require 7.8 TB and 2.3 TB, respectively. Figure 14 shows the accuracy results for the remaining estimators. First, B3.1 extends the NLP scenario from B2.1 with an additional reshape operation. Since this operation is sparsity-preserving, the results are similar to B2.1. Second, B3.4 computes predicted recommendations for known ratings of a selected set of users. We constructed a selection matrix $\mathbf{P}$ to extract the top 10K users according to their number of ratings, and random factors $\mathbf{L}$ and $\mathbf{R}$ with sparsity of 0.95 and 0.85, respectively. The element-wise multiplication has exactly aligned non-zeros and thus, MNC computes the exact sparsity, while MetaAC and the density map fail to recognize this structure. Third, B3.5 applies a boolean predicate of masks $\mathbf{X} \odot ((\mathbf{R} \odot \mathbf{S} + \mathbf{T}) \neq 0)$ to the Mnist1m dataset $\mathbf{X}$. Here, $\mathbf{R}$ is the center mask from B2.5, $\mathbf{S}$ is a random mask with sparsity 0.1 and $\mathbf{T}$ is a data-dependent mask $\mathbf{X} = 255$ for high-intensity pixels. Thus, we select all fully black pixels and a fraction of 0.1 non-zeros in the center area. Although MNC does not yield the exact sparsity, its error of 1.33 is significantly lower then the errors of MetaWC, MetaAC, and density map with 2.13, 2.87, and 2.71, respectively.

All Intermediates for B3.2: The final output of benchmark query B3.2—as used for deferred scaling and shifting—is typically small and dense. However, for matrix chain optimization, the error of all intermediates matters. Disregarding



(a) DMap Relative Errors (b) MNC Relative Errors

Figure 15: Accuracy of All Intermediates for B3.2.

the leaf node reorganizations, we have five matrix products and hence 15 intermediates (i.e., sub chains). Figure 15 compares the accuracy of the density map and MNC for all intermediates. We use Mnist1m as input $\mathbf{X}$ with appended column of ones and left-deep estimation per intermediate. The density map struggles with the special scale-and-shift matrix, resulting in a large final relative error of 98.6. Also, mistakenly estimating $\mathbf{X}\,\mathbf{S}\,\mathbf{B}$ as sparse could lead to a disastrous plan. In contrast, MNC yields the exact sparsity for many intermediates and a very small final error of 1.002. We observed a similar behavior for B3.2 with Cov as input, with errors up to 13,750 for the density map and 2.35 for MNC.

7 RELATED WORK

We review related work from the areas of cardinality estimation, sparsity estimation for matrix products, and advanced optimizations in ML systems that aim to exploit sparsity.

Join Cardinality Estimation: Query cardinality estimation is a well-studied problem, which relates to estimating the number of distinct items or the cardinality of selection, join, and grouping operators. First, distinct item estimation typically relies on sampling [14, 28], or scan-based synopses [8, 31, 36]. Examples for scan-based synopses are HyperLogLog [31] and KMV [8], which have proven accurate but do not directly apply to intermediates. However, the AKMV synopsis (KMV synopsis with counters) allows composing synopses for union, intersection, and difference [8]. Second, join cardinality estimation via sampling [27, 44, 63] faces the challenge that large queries might produce empty results for many distinct items. Other work tackles this challenge via linked synopses [23], correlated sampling [64], or index-based join sampling [45]. Recent work further combines sampling with HyperLogLog sketches for multi-column estimates [22]. Another alternative to sampling is the maintenance of histograms for attributes of base relations [18, 37]. However, to the best of our knowledge, none of these techniques addresses chains of join-group-by pairs (or join-project with set semantics [6]) that could emulate matrix product chains and other operations.

Matrix Product Sparsity Estimation: Apart from the sparsity estimators [5, 10, 16, 39, 49, 65] discussed in Section 2, there is also work on propagating size information and other matrix properties through linear algebra programs. First, similar to constant and type propagation, existing work in Matlab and SystemML propagates constant matrix dimensions according to operation semantics through operator DAGs and conditional control flow [10, 19]. However, these works do not deal with sparsity or rely on conservative worst-case estimates to guarantee memory constraints. Second, based on similar propagation techniques, Sparso [59] also propagates structural properties such as knowledge about symmetric, triangular, and diagonal matrices, which can be exploited by subsequent, data-dependent operations. Third, existing work has already shown that sparsity estimation is crucial for finding good orders during matrix multiplication chain optimization [16, 39]. However, none of these works composes estimates from exact and approximate fractions. Furthermore, the MNC sketch offers a unique balance between good accuracy—especially in the presence of structural properties—and guaranteed low overhead.

Sparsity Estimation in Compressed Sensing: Compressed sensing studies the reconstruction of a sparse matrix $\mathbf{X}$ from few samples. Since many techniques require the sparsity of $\mathbf{X}$ as a parameter, Lopes introduced an estimator of a stable sparsity measure that constitutes a lower bound of the real sparsity [51]. To apply this or similar techniques (e.g., fill estimation for blocked sparse formats [3]) in our context, we would sample cells of $\mathbf{X}_{ij} = \mathbf{A}\,\mathbf{B}$ and compute dot products $\mathbf{A}_i\,\mathbf{B}_j$ to estimate the sparsity s_C. However, similar to $\mathbf{E}_{\text{smpl}}$, this approach does not apply to chains of matrix operations.

Advanced Optimization in ML Systems: Compilation techniques in large-scale ML systems increasingly leverage sparsity for better cost estimation and more efficient runtime plans. Existing techniques include (1) sparsity-exploiting operators in Cumulon [33], SystemML [11], and MatFast [65], (2) automatic operator fusion for sparse operator pipelines [12, 20], and (3) worst-case optimal semi-join reductions [41]. Estimating sparsity for matrix products and element-wise operations also relates to holistic sum-product optimizations [20, 53] of linear algebra programs. Therefore, practical—i.e., accurate and low-overhead—techniques for estimating the sparsity of realistic matrix expressions are widely applicable.

8 CONCLUSIONS

To summarize, after an analysis of existing sparsity estimators, we introduced the very simple, count-based MNC sketch that exploits structural properties for better sparsity estimation. We described estimation algorithms for matrix products, other common operations, and the propagation of sketches for entire DAGs of these operations. Furthermore, we introduced a sparsity estimation benchmark to simplify comparison and foster further improvements. Our experiments have shown that the MNC sketch offers good accuracy—especially in the presence of structural properties—while requiring only modest construction and estimation overhead. In conclusion, the MNC sketch is a versatile tool—which is broadly applicable in modern ML systems—for decisions on formats and preallocation, as well as cost-based, sparsity-aware optimization of DAGs of linear algebra operations. Interesting future work includes (1) MNC sketches in advanced optimizers, (2) confidence intervals, (3) additional operations, and (4) support for distributed matrices and operations.

A EXTENDED SAMPLING-BASED ESTIMATOR

In this appendix, we extend the biased, sampling-based estimator from [65] to yield an unbiased estimator with higher accuracy at similar costs. Furthermore, we briefly describe a more expensive hybrid estimator that relies on hashing and sampling [5], and provide experimental results on how these sampling-based estimators compare.

Unbiased Sampling-based Estimator: The idea is to compute the expected number of non-zeros resulting from element-wise additions of the k sampled outer products as $ml(1 - \prod_{k \in \mathcal{S}}(1 - v_k))$, where $v_k = \text{nnz}(\mathbf{A}_{:k}) \cdot \text{nnz}(\mathbf{B}_{k:})/ml$. We now assume that the remaining $n - |\mathcal{S}|$ outer products are each probabilistically generated according to the empirical distribution of the $|\mathcal{S}|$ observed outer products. That is, the probability that a non-zero appears in any matrix cell in a remaining outer product will be $\overline{v} = (1/|\mathcal{S}|) \sum_{k \in \mathcal{S}} v_k$. Putting all of this together, our sparsity estimate is

$$\hat{s}_C = 1 - (1 - \overline{v})^q \prod_{k \in \mathcal{S}} (1 - v_k), \tag{16}$$

where $q = n - |\mathcal{S}|$. For a chain of matrix products, we take $\text{nnz}(\mathbf{M}^{(j)}_{:k}) = m_j s_j$ when computing s_{j+1}; here s_j is the sparsity estimate for $\mathbf{M}^{(j)}$ and m_j is the number of rows in $\mathbf{M}^{(j)}$. Interestingly, for $n = |\mathcal{S}|$, this approach is equivalent to the MNC fallback case shown in Algorithm 1, line 10.

Hashing- and Sampling-based Estimator [5]: The hash-based estimator [5] extends ideas from KMV (k minimum values) synopses [7, 8]—that maintain the $k = 1/\epsilon^2$ minimum hash values to derive an estimate for the number of distinct items—by the use of pairwise independent hash functions and sampling. Similar to the KMV estimator, this hash-based sparsity estimator is scan-based; it iterates over all columns in $\mathbf{A}_{:i}$ and rows $\mathbf{B}_{i:}$, hashes row and column indexes, samples rows and columns whose hash is below the sample fraction, and maintains a buffer of the k minimum pair hashes. Hence, its time complexity is $O(d + \text{nnz}(\mathbf{A}, \mathbf{B}))$.

Additional Experimental Results: Finally, Table 4 shows the resulting accuracy—in terms of the relative error in $[1, \infty)$—of the extended (i.e., unbiased) estimator as well as the hash-based estimator [5] for all single matrix operations. We observe that the unbiased estimator generally performs very well but is still worse than the more expensive hash-based estimator and much worse than our MNC estimator. Additionally, there are three noteworthy special cases. First, for B1.5, the biased estimator computes the exact result due to its lower-bound assumption, whereas the unbiased estimator fails to recognize the structure and vastly overestimates. Second, the unbiased estimator still struggles to estimate the column projections in B2.2 due to varying sparsity across columns. Third, B2.5 refers to element-wise multiplication, for which we implemented unbiased estimators and thus, there are no differences.

Table 4: Accuracy of Sampling-based Estimators.

Name	Biased [65]	Unbiased	Hash [5]	MNC
B1.1 NLP	84.0	1.55	1.78	**1.0**
B1.2 Scale	53,560	1.01	1.13	**1.0**
B1.3 Perm	92,535	1.27	1.17	**1.0**
B1.4 Outer	INF	INF	1.0	**1.0**
B1.5 Inner	1.0	99,999	INF	**1.0**
B2.1 NLP	44.2	1.60	1.10	**1.0**
B2.2 Project	INF	2.95	1.45	**1.0**
B2.3 CoRefG	54.4	1.80	1.04	**1.17**
B2.4 EmailG	91.8	1.37	1.01	**1.09**
B2.5 Mask	1.76	1.76	N/A	**1.0**

B MULTI-THREADED BITSET ESTIMATOR

The experiments in Section 6 used single-threaded sketch construction and estimation to ensure a fair comparison of estimators and reflect their use during compilation. However, a natural question is if the exact, compute-bound bitset estimator would benefit from multi-threading more than the other estimators (like MNC), which are memory-bandwidth bound. In favor of the dense bitset, we use a dense product of two random 20K × 20K matrices with sparsity 0.99 (see Figure 7(a)). Multi-threading improves the total bitset estimation runtime (construction and estimation) from 128.2 s to 11.7 s (using 12 physical cores), which is a speedup of almost 11x. However, even the single-threaded MNC Basic and MNC still outperform the bitset with 3.2 s and 5.1 s, respectively. Most importantly, the MNC sketch exhibits much better asymptotic behavior for large or very sparse matrices. Unlike the bitset, the MNC runtime is also mostly dominated by sketch construction not estimation, which is important because input sketches can be constructed once and used many times for estimating alternative plans.

C OPTIMIZER INTEGRATION

Although not the focus of this paper, in this section, we present a proof-of-concept integration into SystemML's optimizing compiler. We introduced an additional dynamic rewrite for sparsity-aware matrix multiplication chain optimization, which is still disabled by default.

MMChain Rewrite: Matrix multiplication chain optimization of n matrices $(\mathbf{M}^{(1)}, \mathbf{M}^{(2)}, \ldots, \mathbf{M}^{(n)})$ aims to find the optimal parenthesization of these associative matrix products. The number of alternative plans is computed by the Catalan number $C_{n-1} = (2n - 2)!/(n!(n - 1)!)$. However, this problem can be solved efficiently via dynamic programming [17]. In SystemML, we use a textbook algorithm [17]

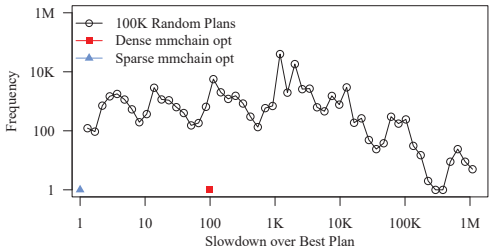


Figure 16: Optimized vs. 100,000 Random Plans.

with cubic time and quadratic space complexity (although an $O(n \log n)$ algorithm exists [32]) due to its simplicity and the rarity of long chains of pure matrix products. This algorithm uses an $n \times n$ cost matrix $\mathbf{C}$ for memoizing optimal costs of subchains, as well as an $n \times n$ matrix $\mathbf{S}$ to keep track of related optimal splits. Our sparsity-aware algorithm is a simple extension by an additional $n \times n$ matrix $\mathbf{E}$ for memoizing sketches of optimal subchains. Instead of computing dense costs, we compute the cost of a subchain by

$$\mathbf{C}_{i,j} = \min_{k \in [i,j-1]} \left(\mathbf{C}_{i,k} + \mathbf{C}_{k+1,j} + \mathbf{E}_{i,k}.\mathbf{h}^c \, \mathbf{E}_{k+1,j}.\mathbf{h}^r \right), \quad (17)$$

because the number of non-zero pairs—computed via the dot product $\mathbf{h}^c \, \mathbf{h}^r$—represents the number of floating point operations of a sparse matrix multiplication (independent of the output sparsity) [16]. For the optimal subplan, we finally perform sketch propagation and store the sketch in $\mathbf{E}_{i,j}$.

Experimental Results: Inspired by recent work on optimizing large join queries [56], we create a matrix product chain of $n = 20$ matrices with dimensions 10, 10^3, 10^4, 10^4, 10^3, 10, 10^4, 1, 10^4, 10^3 (repeated twice), and 1, as well as random sparsity in $[10^{-4}, 1]$ for every third matrix and 0.1 otherwise. Figure 16 plots the distribution of min-normalized costs of 100,000 random plans (of C_{19} = 1,767,263,190 possible plans). Due to uniformly distributed non-zero values, the estimation errors are negligible. We observe that the cost difference between the worst and best plans is more than six orders of magnitude, and that the default dynamic programming algorithm—which is unaware of the sparsity—yields a plan that is 99.1x worse than the best plan. In contrast, our extended rewrite finds the optimal plan. However, this is merely a proof of concept because in practice long chains of matrix products are rare. Instead there is a need for extended algorithms that optimize matrix products and other operations together. We believe this is an interesting direction for future work and our MNC sketch allows computing the necessary sparsity-aware costs.

ACKNOWLEDGMENTS

We thank the authors of the hash-based estimator [5] for providing their source code as well as our anonymous reviewers for their valuable comments.

REFERENCES

[1] Martín Abadi et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*. 265–283.
[2] Ildar Absalyamov, Michael J. Carey, and Vassilis J. Tsotras. 2018. Lightweight Cardinality Estimation in LSM-based Systems. In *SIGMOD*. 841–855.
[3] Peter Ahrens, Helen Xu, and Nicholas Schiefer. 2018. A Fill Estimation Algorithm for Sparse Matrices and Tensors in Blocked Formats. In *IPDPS*. 546–556.
[4] AMiner. 2017. Citation Network Dataset. `static.aminer.cn/lab-datasets/citation/dblp.v10.zip`.
[5] Rasmus Resen Amossen, Andrea Campagna, and Rasmus Pagh. 2014. Better Size Estimation for Sparse Matrix Products. *Algorithmica* 69, 3 (2014), 741–757.
[6] Rasmus Resen Amossen and Rasmus Pagh. 2009. Faster Join-Projects and Sparse Matrix Multiplications. In *ICDT*. 121–126.
[7] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. 2002. Counting Distinct Elements in a Data Stream. In *RANDOM*. 1–10.
[8] Kevin S. Beyer, Peter J. Haas, Berthold Reinwald, Yannis Sismanis, and Rainer Gemulla. 2007. On Synopses for Distinct-Value Estimation Under Multiset Operations. In *SIGMOD*. 199–210.
[9] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Review* 59, 1 (2017).
[10] Matthias Boehm, Douglas R. Burdick, Alexandre V. Evfimievski, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Shirish Tatikonda, and Yuanyuan Tian. 2014. SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. *IEEE Data Eng. Bull.* 37, 3 (2014), 52–62.
[11] Matthias Boehm, Michael Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick Reiss, Prithviraj Sen, Arvind Surve, and Shirish Tatikonda. 2016. SystemML: Declarative Machine Learning on Spark. *PVLDB* 9, 13 (2016), 1425–1436.
[12] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. 2018. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *PVLDB* 11, 12 (2018), 1755–1768.
[13] Léon Bottou. [n. d.]. The infinite MNIST dataset. `http://leon.bottou.org/projects/infimnist`.
[14] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. 2000. Towards Estimation Error Guarantees for Distinct Values. In *PODS*. 268–279.
[15] Edith Cohen. 1994. Estimating the Size of the Transitive Closure in Linear Time. In *FOCS*. 190–200.
[16] Edith Cohen. 1998. Structure Prediction and Computation of Sparse Matrix Products. *J. Comb. Optim.* 2, 4 (1998), 307–332.
[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. MIT Press.
[18] Graham Cormode, Minos N. Garofalakis, Peter J. Haas, and Chris Jermaine. 2012. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases* 4, 1-3 (2012), 1–294.
[19] Luiz A. DeRose. 1996. *Compiler Techniques for MATLAB Programs*. Technical Report.
[20] Tarek Elgamal, Shangyu Luo, Matthias Boehm, Alexandre V. Evfimievski, Shirish Tatikonda, Berthold Reinwald, and Prithviraj Sen. 2017. SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning. In *CIDR*.

[21] Ahmed Elgohary, Matthias Boehm, Peter J. Haas, Frederick R. Reiss, and Berthold Reinwald. 2016. Compressed Linear Algebra for Large-Scale Machine Learning. *PVLDB* 9, 12 (2016), 960–971.

[22] Michael J. Freitag and Thomas Neumann. 2019. Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates. In *CIDR*.

[23] Rainer Gemulla, Philipp Rösch, and Wolfgang Lehner. 2008. Linked Bernoulli Synopses: Sampling along Foreign Keys. In *SSDBM*. 6–23.

[24] John R. Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse Matrices in Matlab: Design and Implementation. *SIAM J. Matrix Anal. Appl.* 13, 1 (1992), 333–356.

[25] Google. [n. d.]. word2vec. `code.google.com/archive/p/word2vec`.

[26] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Lynne Stokes. 1995. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In *VLDB*. 311–322.

[27] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Arun N. Swami. 1996. Selectivity and Cost Estimation for Joins Based on Random Sampling. *J. Comput. Syst. Sci.* 52, 3 (1996), 550–569.

[28] Peter J. Haas and Lynne Stokes. 1998. Estimating the Number of Classes in a Finite Population. *J. Amer. Statist. Assoc.* 93, 444 (1998), 1475–1487.

[29] Hazar Harmouch and Felix Naumann. 2017. Cardinality Estimation: An Experimental Survey. *PVLDB* 11, 4 (2017), 499–512.

[30] Ruining He and Julian McAuley. 2016. Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering. In *WWW*. 507–517.

[31] Stefan Heule, Marc Nunkesser, and Alexander Hall. 2013. HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *EDBT*. 683–692.

[32] T. C. Hu and M. T. Shing. 1984. Computation of Matrix Chain Products. Part II. *SIAM J. Comput.* 13, 2 (1984), 228–251.

[33] Botong Huang, Shivnath Babu, and Jun Yang. 2013. Cumulon: Optimizing Statistical Data Analysis in the Cloud. In *SIGMOD*. 1–12.

[34] Botong Huang, Matthias Boehm, Yuanyuan Tian, Berthold Reinwald, Shirish Tatikonda, and Frederick R. Reiss. 2015. Resource Elasticity for Large-Scale Machine Learning. In *SIGMOD*. 137–152.

[35] Yannis E. Ioannidis and Stavros Christodoulakis. 1991. On the Propagation of Errors in the Size of Join Results. In *SIGMOD*. 268–277.

[36] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. 2010. An optimal algorithm for the distinct elements problem. In *PODS*. 41–52.

[37] Carl-Christian Kanne and Guido Moerkotte. 2010. Histograms Reloaded: The Merits of Bucket Diversity. In *SIGMOD*. 663–674.

[38] Jeremy Kepner and John Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra*. SIAM.

[39] David Kernert, Frank Köhler, and Wolfgang Lehner. 2015. SpMacho - Optimizing Sparse Linear Algebra Expressions with Probabilistic Density Estimation. In *EDBT*. 289–300.

[40] David Kernert, Wolfgang Lehner, and Frank Köhler. 2016. Topology-Aware Optimization of Big Sparse Matrices and Matrix Multiplications on Main-Memory Systems. In *ICDE*. 823–834.

[41] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *PODS*. 13–28.

[42] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *EMNLP*. 1746–1751.

[43] Arun Kumar, Matthias Boehm, and Jun Yang. 2017. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *SIGMOD*. 1717–1722.

[44] Per-Åke Larson, Wolfgang Lehner, Jingren Zhou, and Peter Zabback. 2007. Cardinality estimation using sample views with quality assurance. In *SIGMOD*. 175–186.

[45] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *CIDR*.

[46] J. Leskovec, J. Kleinberg, and C. Faloutsos. [n. d.]. SuiteSparse Matrix Collection: email-EuAll. `sparse.tamu.edu/SNAP/email-EuAll`.

[47] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. 2007. Graph evolution: Densification and shrinking diameters. *TKDD* 1, 1 (2007).

[48] M. Lichman. [n. d.]. UCI Machine Learning Repository: Covertype. `https://archive.ics.uci.edu/ ml/datasets/Covertype`.

[49] Weifeng Liu and Brian Vinter. 2014. An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data. In *IPDPS*. 370–381.

[50] Gaëlle Loosli, Stéphane Canu, and Léon Bottou. 2007. Training Invariant Support Vector Machines using Selective Sampling. In *Large Scale Kernel Machines*, Léon Bottou, Olivier Chapelle, Dennis DeCoste, and Jason Weston (Eds.). MIT Press, Cambridge, MA., 301–320.

[51] Miles Lopes. 2013. Estimating Unknown Sparsity in Compressed Sensing. In *ICML*. 217–225.

[52] Julian McAuley. [n. d.]. Amazon Product Data - Books. `jmcauley.ucsd.edu/data/amazon`.

[53] Vijay Menon and Keshav Pingali. 1999. High-Level Semantic Optimization of Numerical Codes. In *ICS*. 434–443.

[54] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* (2013).

[55] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *NIPS*. 3111–3119.

[56] Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *SIGMOD*. 677–692.

[57] Milos Nikolic, Mohammed Elseidy, and Christoph Koch. 2014. LINVIEW: incremental view maintenance for complex analytical queries. In *SIGMOD*. 253–264.

[58] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[59] Hongbo Rong, Jongsoo Park, Lingxiang Xiang, Todd A. Anderson, and Mikhail Smelyanskiy. 2016. Sparso: Context-driven Optimizations of Sparse Linear Algebra. In *PACT*. 247–259.

[60] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June Paul Hsu, and Kuansan Wang. 2015. An Overview of Microsoft Academic Service (MAS) and Applications. In *WWW - Companion Volume*. 243–246.

[61] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. 2011. The Architecture of SciDB. In *SSDBM*. 1–16.

[62] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. 2008. ArnetMiner: Extraction and Mining of Academic Social Networks. In *SIGKDD*. 990–998.

[63] Wentao Wu, Jeffrey F. Naughton, and Harneet Singh. 2016. Sampling-Based Query Re-Optimization. In *SIGMOD*. 1721–1736.

[64] Feng Yu, Wen-Chi Hou, Cheng Luo, Dunren Che, and Mengxia Zhu. 2013. CS2: a new database synopsis for query estimation. In *SIGMOD*. 469–480.

[65] Yongyang Yu, MingJie Tang, Walid G. Aref, Qutaibah M. Malluhi, Mostafa M. Abbas, and Mourad Ouzzani. 2017. In-Memory Distributed Matrix Computation Processing and Optimization. In *ICDE*.

[66] Reza Bosagh Zadeh, Xiangrui Meng, Alexander Ulanov, Burak Yavuz, Li Pu, Shivaram Venkataraman, Evan R. Sparks, Aaron Staple, and Matei Zaharia. 2016. Matrix Computations and Optimization in Apache Spark. In *SIGKDD*. 31–38.