SCIENCE
PASSION
TECHNOLOGY

Architecture of ML Systems
03 Size Inference, Rewrites, and Operator Selection

Matthias Boehm

Graz University of Technology, Austria
Computer Science and Biomedical Engineering
Institute of Interactive Systems and Data Science
BMVIT endowed chair for Data Management

Last update: Mar 29, 2019

Announcements/Org

- **#1 Modified Course Logistics**
  - **5 ECTS** (lectures+exam, and project),
    ➔ pick a **(1) programming project**, or
    (2) survey / experimental analysis project

- **#2 Programming/Analysis Projects**
  - **Apr 05:** Project selection
  - Discussion individual projects (first come, first served)

Agenda

- **Compilation Overview**
- **Size Inference and Cost Estimation**
- **Rewrites and Operator Selection**

Compilation Overview

Recap: Linear Algebra Systems

- **Comparison Query Optimization**
  - **Rule- and cost-based rewrites and operator ordering**
  - **Physical operator selection and query compilation**
  - Linear algebra / other ML operators, DAGs, control flow, sparse/dense formats

DB

PL **HPC**

Compilers for Large-scale ML

- **#1 Interpretation** (operation at-a-time)
  - Examples: **R**, **PyTorch**, **Morpheus** [PVLDB'17]
- **#2 Lazy Expression Compilation** (DAG at-a-time)
  - Examples: **RIOT** [CIDR'09], **Mahout Samsara** [MLSystems'16]
  - Examples w/ control structures: **Weld** [CIDR'17], **OptiML** [ICML'11], **Emma** [SIGMOD'15]
- **#3 Program Compilation** (entire program)
  - Examples: **SystemML** [PVLDB'16], **Julia**, **Cumulon** [SIGMOD'13], **Tupleware** [PVLDB'15]

Optimization Scope

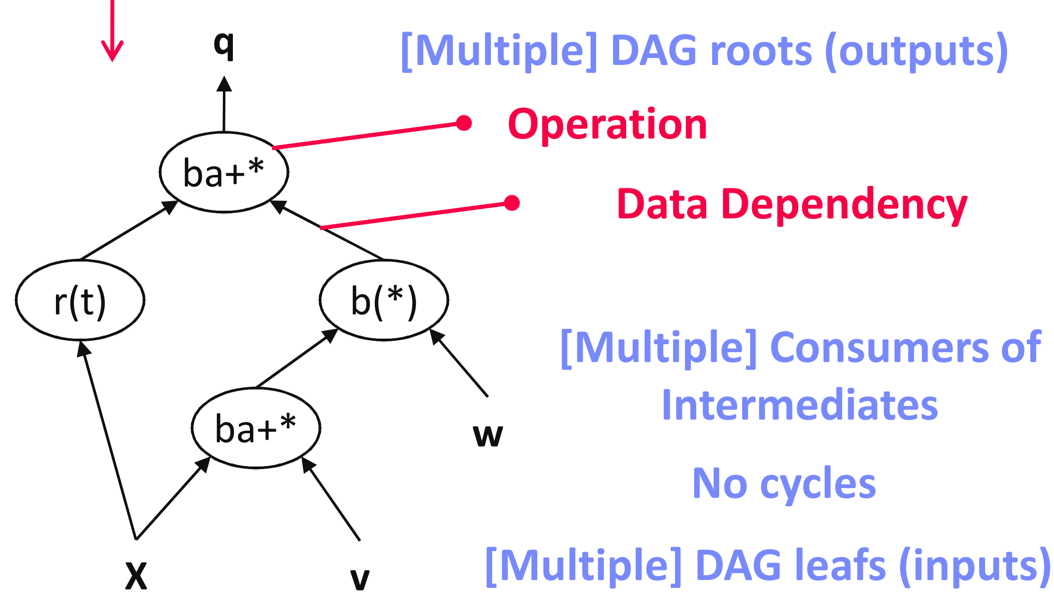
```
X = read($1); # n x m matrix
y = read($2); # n x 1 vector
maxi = 50; lambda = 0.001;
intercept = $3;
...
r = -(t(X) %*% y);
norm_r2 = sum(r * r); p = -r;
w = matrix(0, ncol(X), 1); i = 0;
while(i<maxi & norm_r2>norm_r2_trgt)
{
   q = (t(X) %*% X %*% p)+lambda*p;
   alpha = norm_r2 / sum(p * q);
   w = w + alpha * p;
   old_norm_r2 = norm_r2;
   r = r + alpha * q;
   norm_r2 = sum(r * r);
   beta = norm_r2 / old_norm_r2;
   p = -r + beta * p; i = i + 1;
}
write(w, $4, format="text");
```

ML Program Compilation

- **Script:**

```
while(...) {
   q = t(X) %*% (w * (X %*% v)) ...
}
```

- **Operator DAG**
  - a.k.a. “graph”
  - a.k.a. intermediate representation (IR)

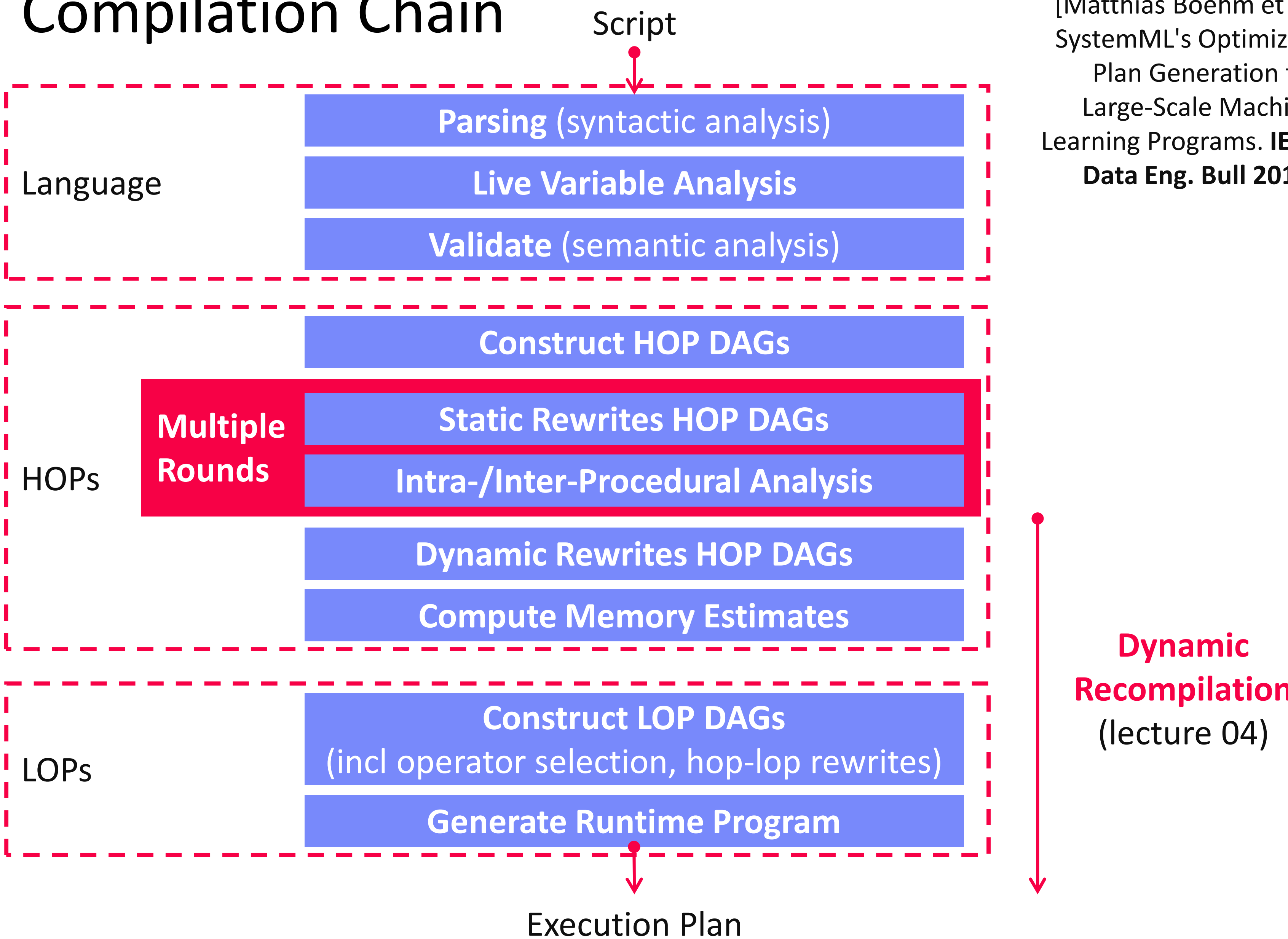


- **Runtime Plan**
  - Compiled runtime plans Interpreted plans

```
SPARK mapmmchain X.MATRIX.DOUBLE w.MATRIX.DOUBLE
v.MATRIX.DOUBLE _mVar4.MATRIX.DOUBLE XtwXv
```

Compilation Chain

[Matthias Boehm et al: SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. **IEEE Data Eng. Bull 2014**]



Recap: Basic HOP and LOP DAG Compilation

LinregDS (Direct Solve)

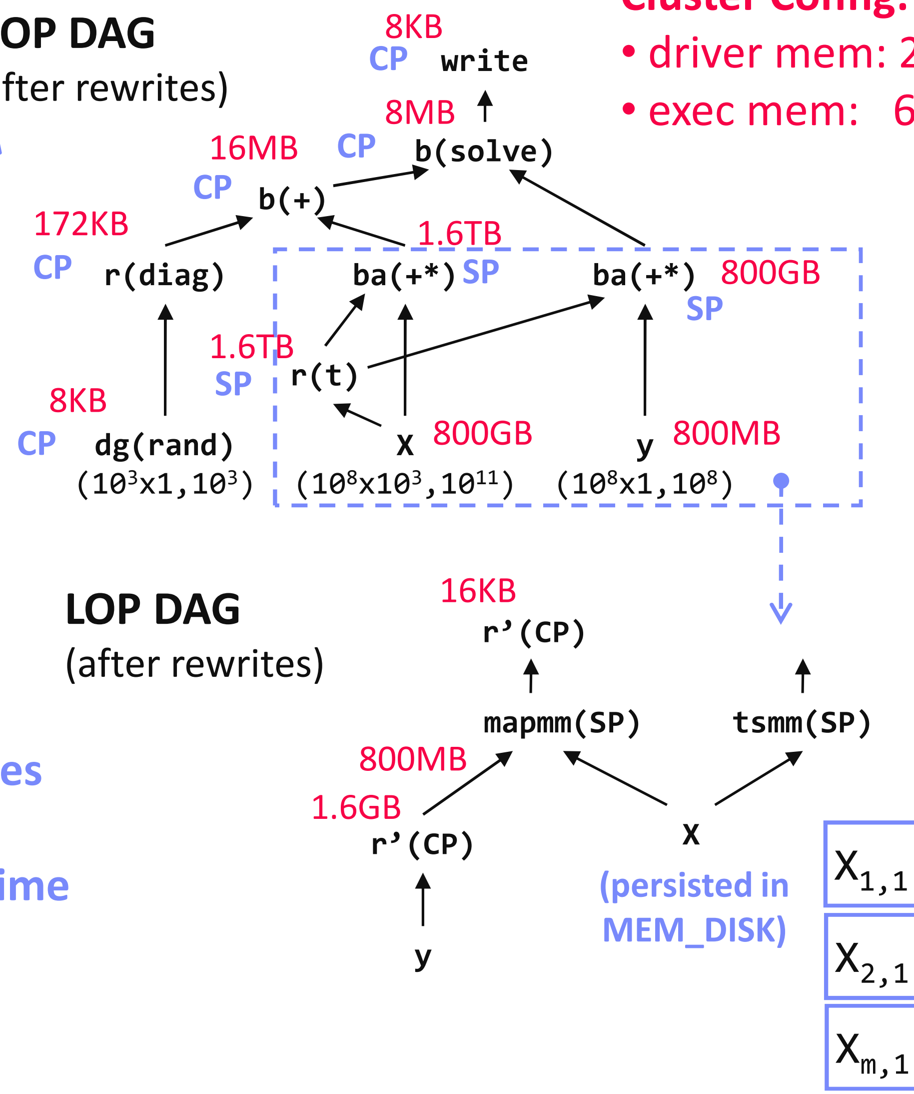
```
X = read($1);
y = read($2);
intercept = $3;
lambda = 0.001;
...
if( intercept == 1 ) {
  ones = matrix(1, nrow(X), 1);
  X = append(X, ones);
}
I = matrix(1, ncol(X), 1);
A = t(X) %*% X + diag(I)*lambda;
b = t(X) %*% y;
beta = solve(A, b);
...
write(beta, $4);
```

Scenario:
X: $10^8 \times 10^3$, 10^{11}
y: $10^8 \times 1$, 10^8

Cluster Config:
- driver mem: 20 GB
- exec mem: 60 GB

HOP DAG (after rewrites)

LOP DAG (after rewrites)



➔ **Hybrid Runtime Plans:**
- **Size propagation / memory estimates**
- **Integrated CP / Spark runtime**
- **Dynamic recompilation during runtime**

➔ **Distributed Matrices**
- **Fixed-size (squared) matrix blocks**
- **Data-parallel operations**

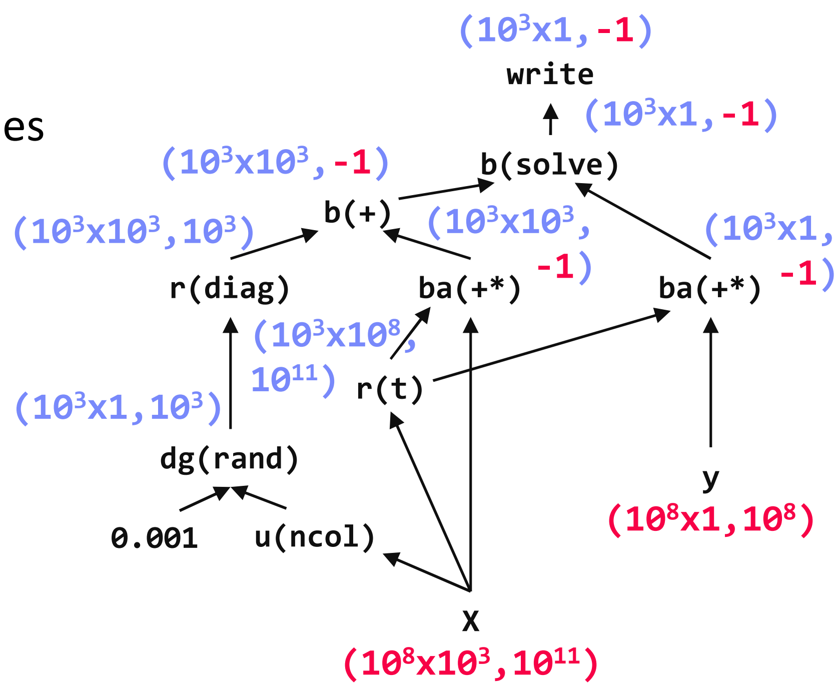
Size Inference and Cost Estimation

Constant and Size Propagation

- **Size Information**
  - Dimensions (#rows, #columns)
  - Sparsity (#nnz/(#rows * #columns))
  - ➔ **memory estimates and costs**

```
X = read($1);
y = read($2);
I = matrix(0.001, ncol(X), 1);
A = t(X) %*% X + diag(I);
b = t(X) %*% y;
beta = solve(A, b);
```

- **DAG-level Size Propagation**
  - **Input:** Size information for leaf nodes
  - **Output:** size information for all operators, -1 if still unknown
  - **Propagation based on operation semantics** (single bottom-up pass over DAG)



Constant and Size Propagation, cont.

- **Constant Propagation**
  - Relies on live variable analysis
  - Propagate constant literals into read-only statement blocks

- **Program-level Size Propagation**
  - Relies on **constant propagation** and **DAG-level size propagation**
  - **Propagate size information across conditional control flow:** size in leafs, DAG-level prop, extract roots
  - **if:** reconcile if and else branch outputs
  - **while/for:** reconcile pre and post loop, reset if pre/post different

```
X = read($1); # n x m matrix
y = read($2); # n x 1 vector
maxi = 50; lambda = 0.001;
if(...){ }
r = -(t(X) %*% y);
r2 = sum(r * r);
p = -r;                          # m x 1
w = matrix(0, ncol(X), 1);       # m x 1
i = 0;
while(i<maxi & r2>r2_trgt) {
   q = (t(X) %*% X %*% p)+lambda*p;
   alpha = norm_r2 / sum(p * q);
   w = w + alpha * p;            # m x 1
   old_norm_r2 = norm_r2;
   r = r + alpha * q;
   r2 = sum(r * r);
   beta = norm_r2 / old_norm_r2;
   p = -r + beta * p;            # m x 1
   i = i + 1;
}
write(w, $4, format="text");
```

Inter-Procedural Analysis

- **Intra/Inter-Procedural Analysis (IPA)**
  - Integrates all size propagation techniques (**DAG+program**, **size+constants**)
  - Intra-function and inter-function size propagation (**called once**, **consistent sizes**, **consistent literals**)

```
X = read($X1)
X = foo(X);
if( $X2 != “ ” ) {
  X2 = cbind(X,
    matrix(1,n,1));
  X2 = foo(X2);
}...
```

1M x 1

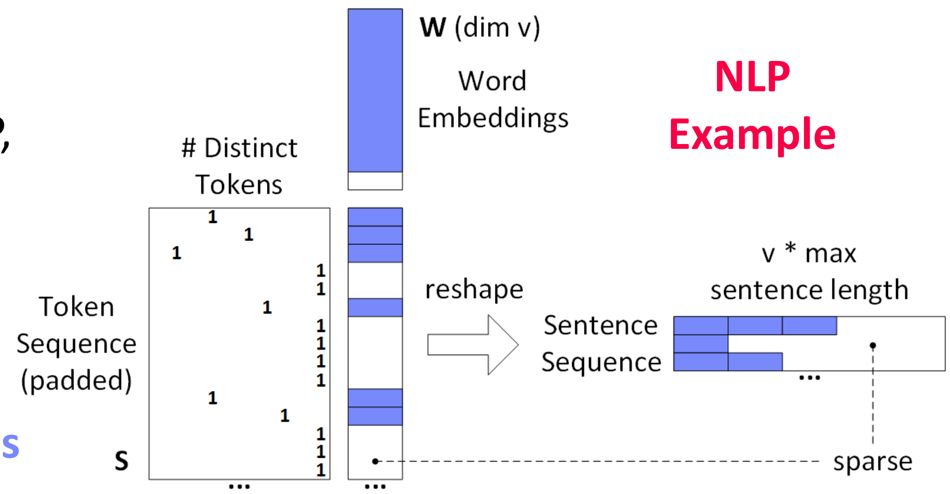
1M x 2

```
foo = function (Matrix[Double] A)
    return (Matrix[Double] B)
{
  B = A - colSums(A);
  if( sum(B!=B)>0 )
    print(“NaNs encountered.”);
}
```

- **Additional IPA Passes (selection)**
  - **Inline functions** (single statement block, small)
  - **Dead code elimination** and simplification rewrites
  - Remove unused functions & flag functions for recompile-once

Sparsity Estimation Overview

- **Motivation**
  - Sparse input matrices from NLP, graph analytics, recommender systems, scientific computing
  - Sparse intermediates (selections, dropout)
  - Selection/permutation matrices

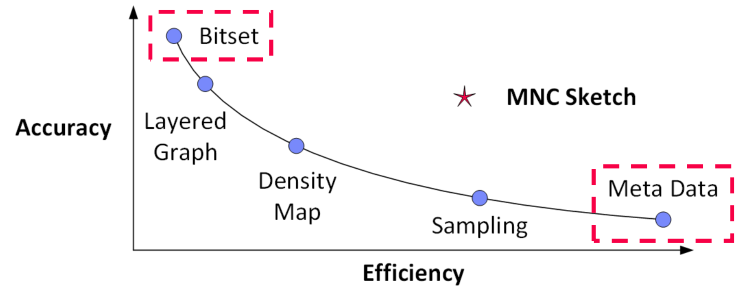


- **Problem Definition**
  - Sparsity estimates used for **format decisions**, **output allocation**, **cost estimates**
  - Matrix A with sparsity s_A = nnz(A)/(mn) and matrix B with s_B = nnz(B)/(nl)
  - Estimate sparsity s_C of matrix product C = A B; d=max(m,n,l)
  - Assumptions (Boolean matrix product)
    - **A1:** No cancellation errors (round of errors)
    - **A2:** No not-a-number (NaN)

Sparsity Estimation – Naïve Estimators

- **Average-case Estimator (meta data)**
  - Computes output sparsity based on s_A and s_B (e.g., SystemML, SpMachO)
  - Assumes uniform nnz distribution
  - $\hat{s}_C = 1 - (1 - s_A \cdot s_B)^n$,

O(1)
O(1)



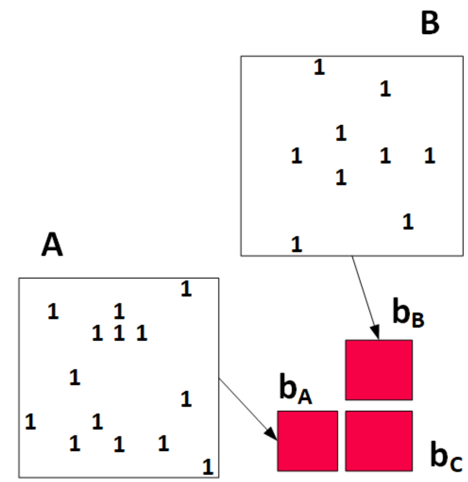
- **Worst-case Estimator (meta data)**
  - Computes output sparsity based on sA and sB (e.g., SystemML)
  - Assumes worst-case scenario (upper bound)

O(1)
O(1)

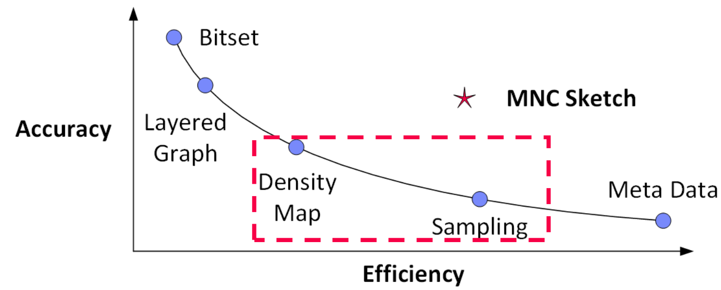
$$\hat{s}_C = \min(1, \text{nnz}(\mathbf{A})/m) \cdot \min(1, \text{nnz}(\mathbf{B})/l)$$
$$= \min(1, s_A \cdot n) \cdot \min(1, s_B \cdot n),$$

- **Bitset Estimator**
  - Constructs Boolean matrices and performs an exact Boolean matrix multiply (e.g., cuSPARSE, MKL, SciDB)
  - $s_C = \hat{s}_C = \text{bitcount}(\mathbf{b}_C)/(ml)$,

O(mn+nl +ml)
O(mnl)

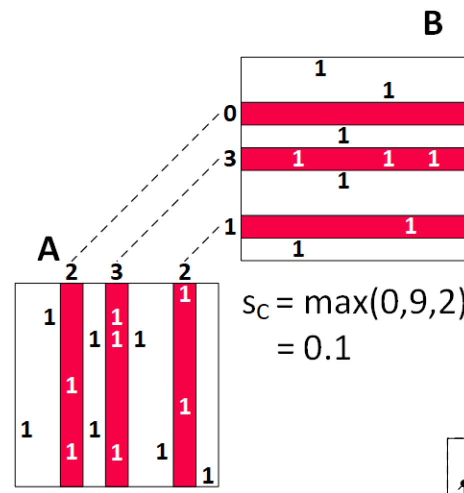


Sparsity Estimation – Sampling and Density Map



- **Sampling-based Estimator**
  - Takes a sample S of aligned columns in A and rows in B (e.g., MatFast)
  - Estimates single matrix product via no-collisions assumption (lower bound)
  - Biased: $\hat{s}_C = \max_{k \in \mathcal{S}} (\text{nnz}(\mathbf{A}_{:k}) \cdot \text{nnz}(\mathbf{B}_{k:}))/(ml).$
  - (Unbiased: $\hat{s}_C = 1 - (1 - \overline{v})^q \prod_{k \in \mathcal{S}} (1 - v_k),$)

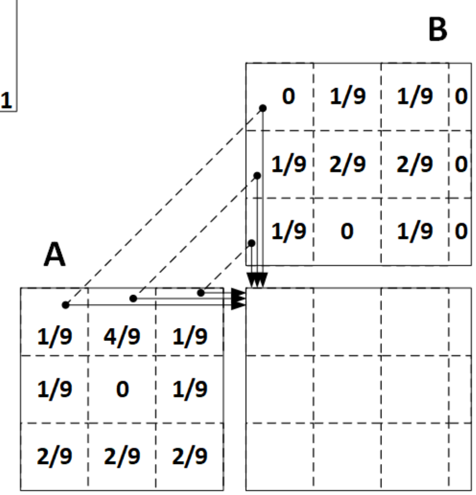
O(|S|)
O(|S|(m+l))



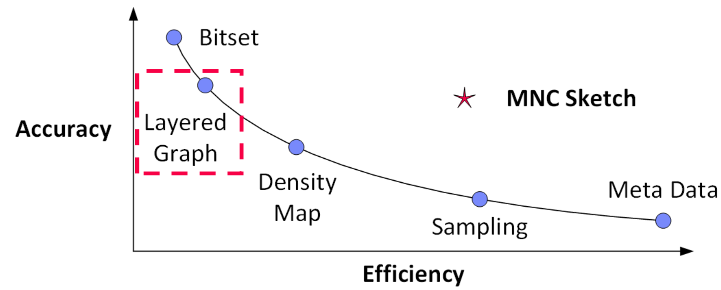
s_C = max(0,9,2)/90
= 0.1

- **DensityMap Estimator**
  - Creates density map of squared block size b=256 (e.g., SPMachO)
  - Estimate chains via average-case estimates
  - $\text{dm}_{C_{ij}} = \oplus_{k=1}^{n/b} \text{E}_{\text{ac}}(\text{dm}_{A_{ik}}, \text{dm}_{B_{kj}})$
    with $s_{A \oplus B} = s_A + s_B - s_A s_B,$

$O(mn/b^2 + nl/b^2 + ml/b^2)$
$O(mnl/b^3)$



Sparsity Estimation – Layered Graph



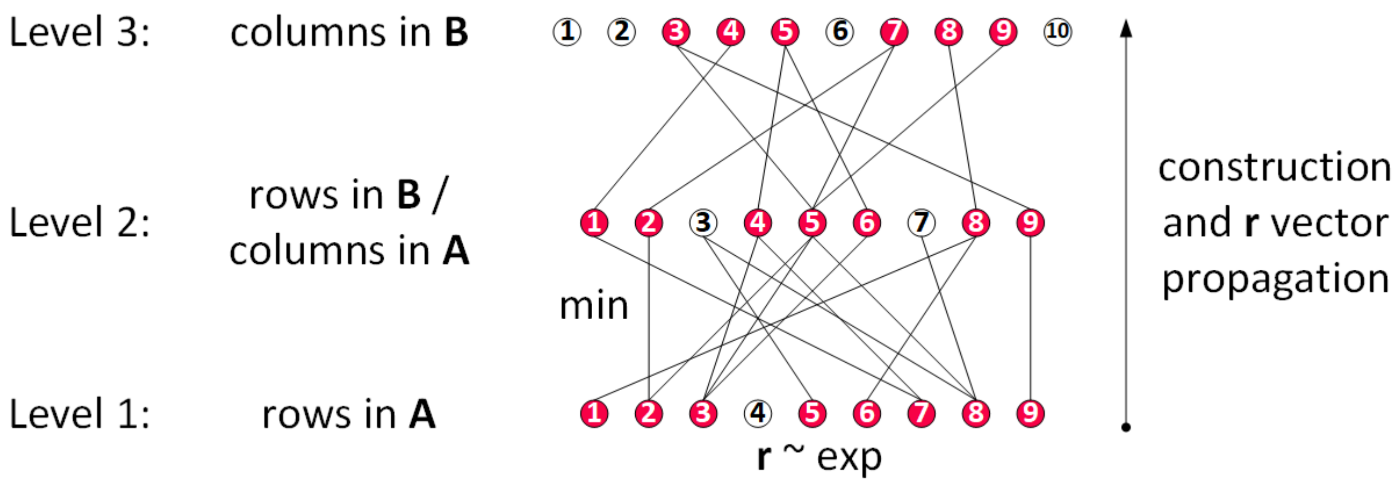
- **Layered Graph**

O(rd +nnz(A,B))

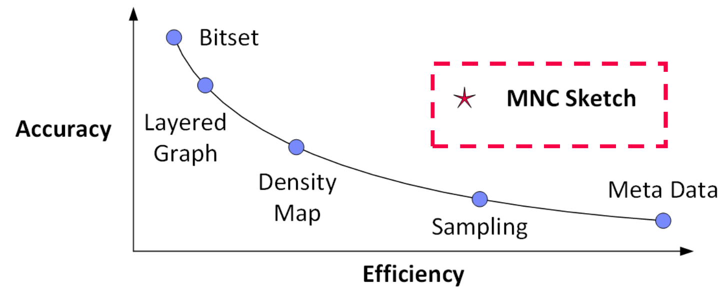
O(r(d +nnz(A,B)))

  - Construct layered graph for mm chain, where nodes represents rows/columns, and edges represent non-zeros
  - Assign vector r (variable size) to leafs, propagate via min(r1,…,rn), and estimate column counts as

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



Sparsity Estimation – MNC (Matrix Non-zero Count)



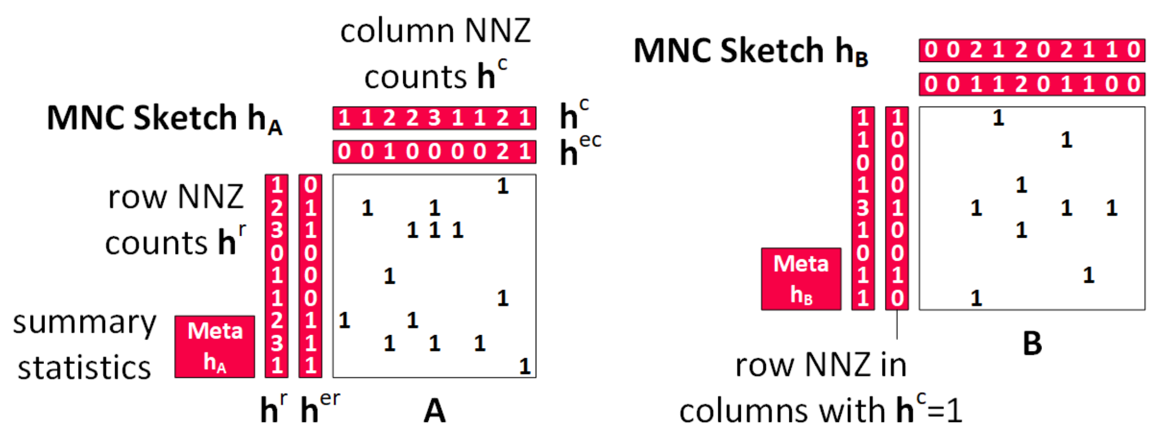
- **MNC Estimator (SystemML, SystemDS)**
  - Create MNC sketch for inputs A and B
  - Exact nnz estimates if structure

    $$s_C \equiv \hat{s}_C = \mathbf{h}^c_A \, \mathbf{h}^r_B / (ml) \; \textit{if} \; \max(\mathbf{h}^r_A) \leq 1 \vee \max(\mathbf{h}^c_B) \leq 1.$$
  - Partial exact/approximate nnz estimates, or fallbacks otherwise

O(d)

O(d +nnz(A,B))

[Johanna Sommer, Matthias Boehm, Alexandre V. Evfimievski, Berthold Reinwald, Peter J. Haas: MNC: Structure-Exploiting Sparsity Estimation for Matrix Expressions. **SIGMOD 2019**]



  - Support for other operations (reorganizations, elementwise ops)
  - Propagate sketches via sparsity estimation and scaling of input sketches

Memory Estimates and Costing

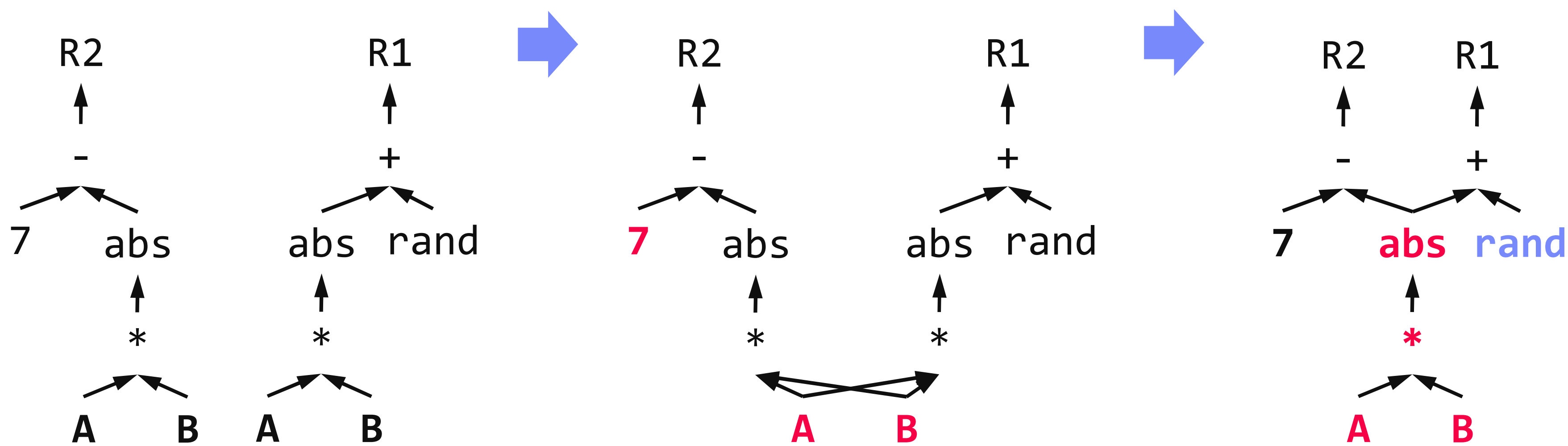
- **Memory Estimates**
  - **Matrix memory estimate :=** based on the dimensions and sparsity, decide the format (sparse, dense) and estimate the size in memory
  - **Operation memory estimate :=** input, intermediates, output
  - **Worst-case sparsity estimates** (**upper bound**)
- **Costing at Logical vs Physical Level**
  - Costing at physical level takes physical ops and rewrites into account but is much more costly
- **Costing Operators vs Plans**
  - Costing plans requires heuristics for **# iterations**, **branches** in general
- **Analytical vs Trained Cost Models**
  - Analytical: **estimate I/O and compute workload**
  - Training: **build regression models** for individual ops

Rewrites and Operator Selection

Traditional PL Rewrites

```
R1 = 7 – abs(A * B)
R2 = abs(A * B) + rand()
```

- **#1 Common Subexpression Elimination (CSE)**
  - **Step 1:** Collect and **replace leaf nodes** (variable reads and literals)
  - **Step 2:** recursively **remove CSEs bottom-up** starting at the leafs by merging nodes with same inputs (**beware non-determinism**)



Topic #10 Common Subexpression Elimination & Constant Folding

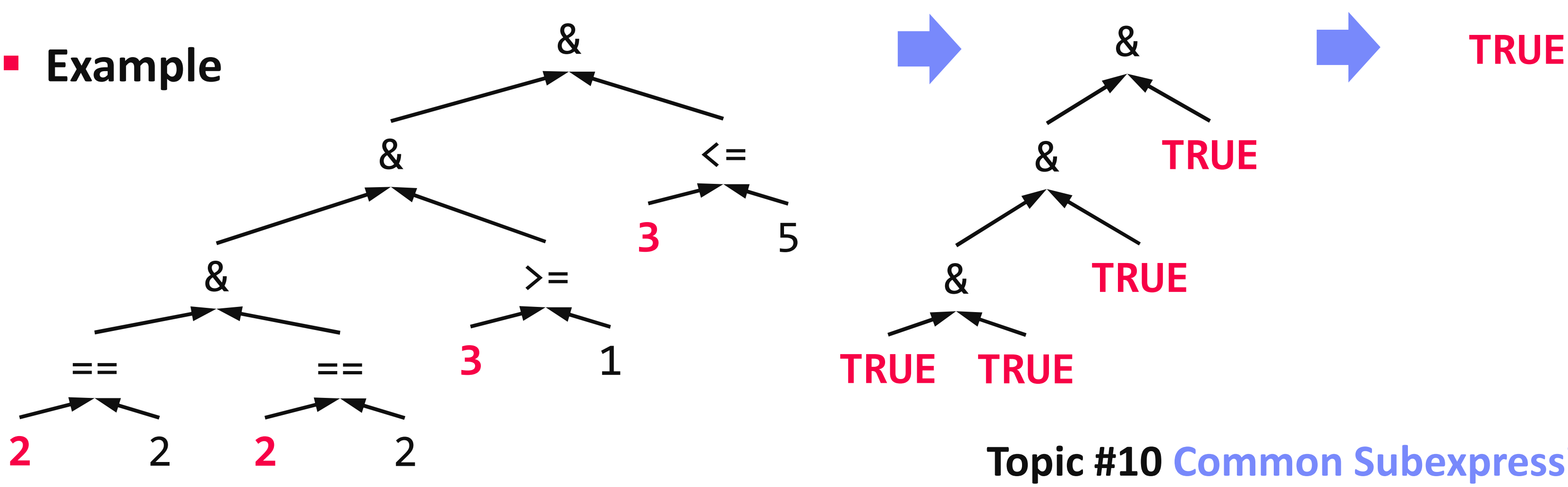
Traditional PL Rewrites, cont.

- **#2 Constant Folding**
  - Applied **after constant propagation**
  - Fold sub-DAGs over literals into a single literal
  - Handling of one-side constants
  - Approach: **recursively** compile and **execute runtime instructions**

```
ncol_y == 2 & dist_type == 2
  & link_type >= 1 & link_type <= 5
```

```
2 == 2 & 2 == 2 & 3 >= 1 & 3 <= 5
```

- **Example**



Topic #10 Common Subexpression Elimination & Constant Folding

Traditional PL Rewrites, cont.

- **#3 Branch Removal**
  - Applied after **constant propagation** and **constant folding**
  - **True predicate:** replace if statement block with if-body blocks
  - **False predicate:** replace if statement block with else-body block, or remove

- **#4 Merge of Statement Blocks**
  - **Merge sequences of unconditional blocks** (s1,s2) into a single block
  - Connect matching DAG roots of s1 with DAG inputs of s2

LinregDS (Direct Solve)

```
X = read($1);
y = read($2);
intercept = 0;
lambda = 0.001;
...        FALSE
if( intercept == 1 ) {
  ones = matrix(1, nrow(X), 1);
  X = cbind(X, ones);
}
I = matrix(1, ncol(X), 1);
A = t(X) %*% X + diag(I)*lambda;
b = t(X) %*% y;
beta = solve(A, b);
...
write(beta, $4);
```

Vectorization and Incremental Computation

- **Loop Transformations** (e.g., OptiML, SystemML)
  - **Loop vectorization**
  - Loop hoisting

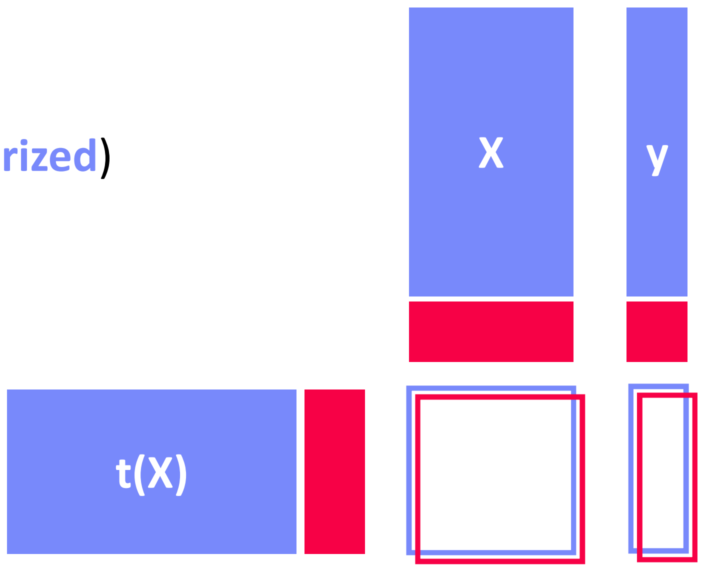
```
for(i in a:b)
   X[i,1] = Y[i,2] + Z[i,1]
```

→

```
X[a:b,1] = Y[a:b,2] + Z[a:b,1]
```

- **Incremental Computations**
  - **Delta update rules** (e.g., LINVIEW, factorized)
  - Incremental iterations (e.g., Flink)

```
A = t(X) %*% X + t(ΔX) %*% ΔX
b = t(X) %*% y + t(ΔX) %*% Δy
```



Update-in-place

- **Example: Cumulative Aggregate via Strawman Scripts**
  - **But:** R, Julia, Matlab, SystemML, NumPy all provide cumsum(X), etc

```
cumsumN2 = function(Matrix[Double] A)
  return(Matrix[Double] B)
{
  B = A; csums = matrix(0,1,ncol(A));
  for( i in 1:nrow(A) ) {
    csums = csums + A[i,];
    B[i,] = csums;
  }
}
```

copy-on-write → O(n^2)

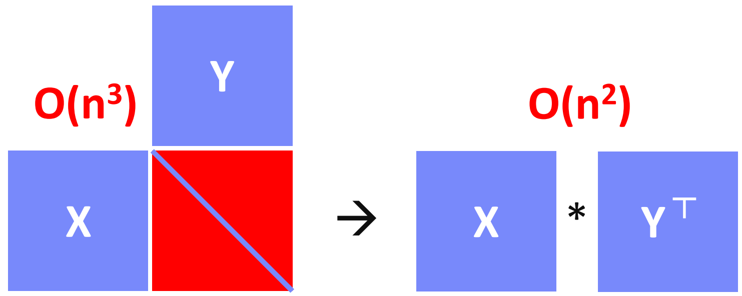
```
cumsumNlogN = function(Matrix[Double] A)
  return(Matrix[Double] B)
{
  B = A; m = nrow(A); k = 1;
  while( k < m ) {
    B[(k+1):m,] = B[(k+1):m,] + B[1:(m-k),];
    k = 2 * k;
  }
}
```

→ O(n log n)

- **Update in place** (w/ O(n))
  - **SystemML:** via rewrites (**why do the above scripts apply?**)
  - **R:** via reference counting
  - **Julia:** by default, otherwise explicit **B = copy(A)** necessary

Static and Dynamic Simplification Rewrites

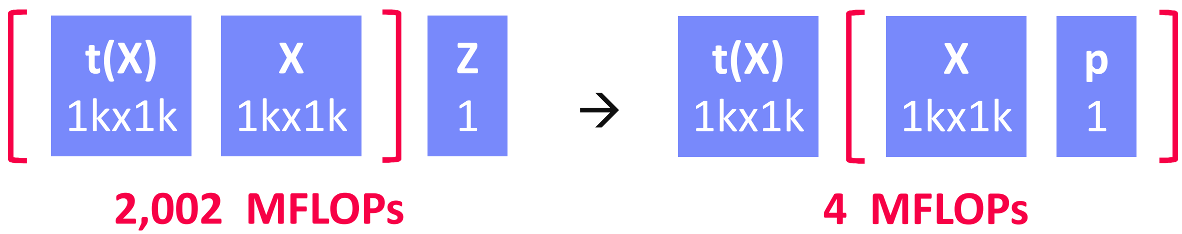
- **Examples of Static Rewrites**
  - `t(X) %*% y → t(t(y) %*% X)`
  - `trace(X%*%Y) → sum(X*t(Y))`
  - `sum(X+Y) → sum(X)+sum(Y)`
  - `(X%*%Y)[7,3] → X[7,]%*%Y[,3]`
  - `sum(t(X)) → sum(X)`
  - `rand()*7 → rand(,min=0,max=7)`
  - `sum(lambda*X) → lambda * sum(X);`



- **Examples of Dynamic Rewrites**
  - `X[a:b,c:d]=Y → X = Y iff dims(X)=dims(Y)`
  - `(...) * X → matrix(0, nrow(X), ncol(X)) iff nnz(X)=0`
  - `sum(X^2) → t(X)%*%X; rowSums(X) → X iff ncol(X)=1`
  - `sum(X%*%Y) → sum(t(colSums(X))*rowSums(Y)) iff ncol(X)>t`

Matrix Multiplication Chain Optimization

- **Optimization Problem**
  - Matrix multiplication chain of n matrices $M_1, M_2, \ldots M_n$ (associative)
  - Optimal parenthesization of the product $M_1 M_2 \ldots M_n$



Size propagation and sparsity estimation

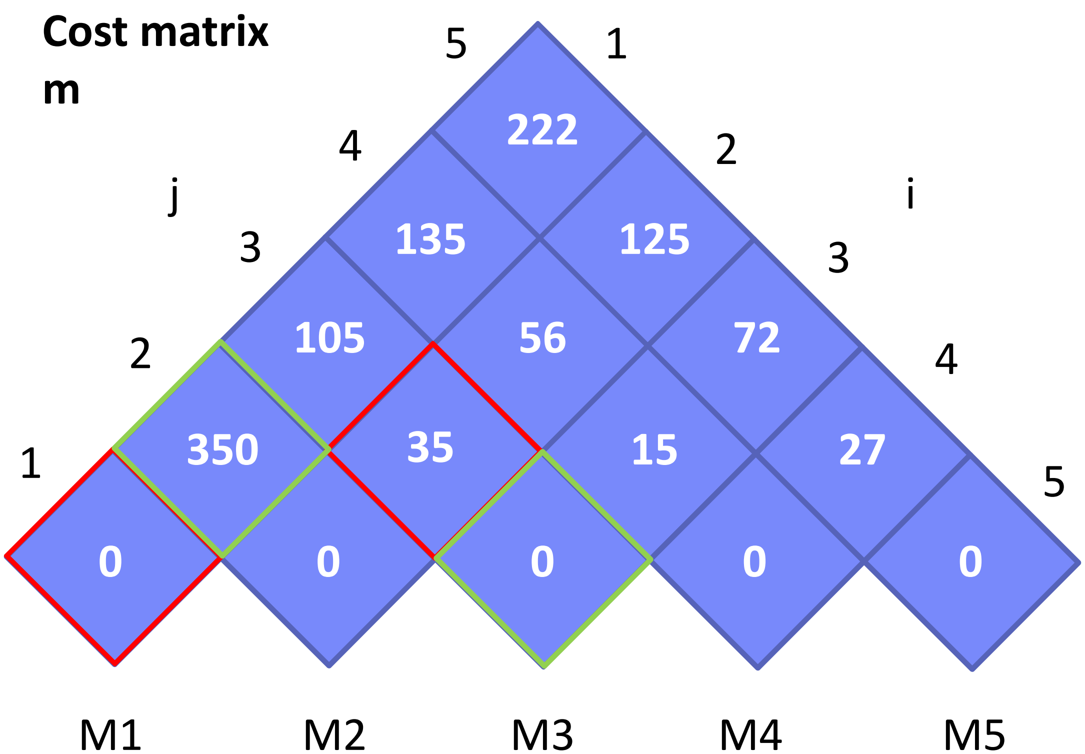
- **Search Space Characteristics**
  - Naïve exhaustive: Catalan numbers → $\Omega(4^n / n^{3/2})$)
  - DP applies: (1) optimal substructure, (2) overlapping subproblems
  - Textbook DP algorithm: $\Theta(n^3)$ time, $\Theta(n^2)$ space
    - Examples: SystemML '14, RIOT ('09 I/O costs), SpMachO ('15 sparsity)
  - Best known algorithm ('81): $O(n \log n)$

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

n	C_{n-1}
5	14
10	4,862
15	2,674,440
20	1,767,263,190
25	1,289,904,147,324

Matrix Multiplication Chain Optimization, cont.

M1	M2	M3	M4	M5
10x7	7x5	5x1	1x3	3x9



m[1,3] = min(
m[1,1] + m[2,3] + p1p2p4,
m[1,2] + m[3,3] + p1p3p4)

= min(
0 + 35 + 10*7*1,
350 + 0 + 10*5*1)

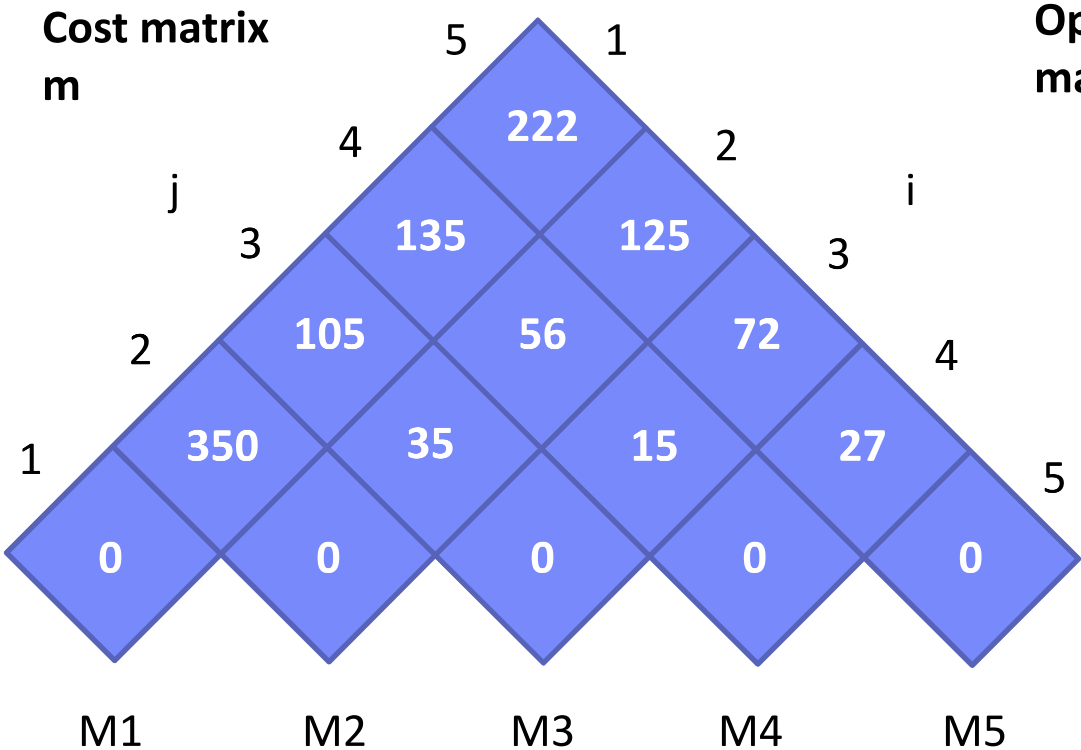
= min(
105,
400)

[T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: Introduction to Algorithms, Third Edition, **The MIT Press**, pages 370-377, 2009]

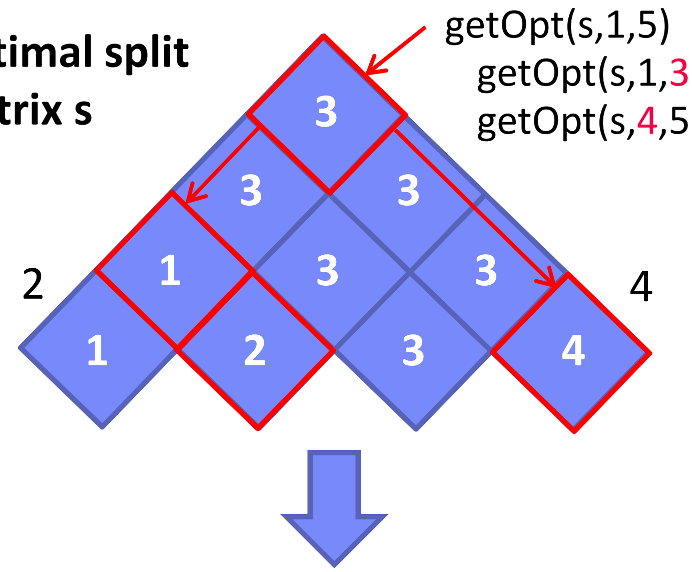
Matrix Multiplication Chain Optimization, cont.

M1	M2	M3	M4	M5
10x7	7x5	5x1	1x3	3x9

Cost matrix m



Optimal split matrix s



(M1 M2 M3 M4 M5)

((M1 M2 M3) (M4 M5))

((M1 (M2 M3)) (M4 M5))

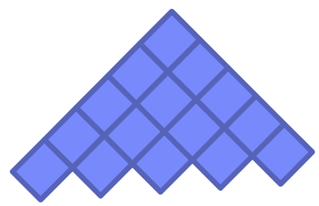
→ ((M1 (M2 M3)) (M4 M5))

→ Open questions: DAGs; other operations, sparsity joint opt w/ rewrites, CSE, fusion, and physical operators

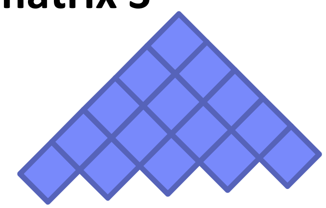
Matrix Multiplication Chain Optimization, cont.

- **Sparsity-aware mmchain Opt**
  - Additional n x n sketch matrix e

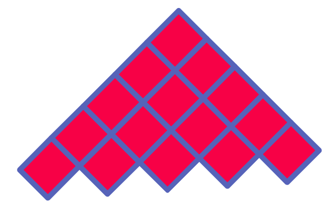
Cost matrix M



Optimal split matrix S

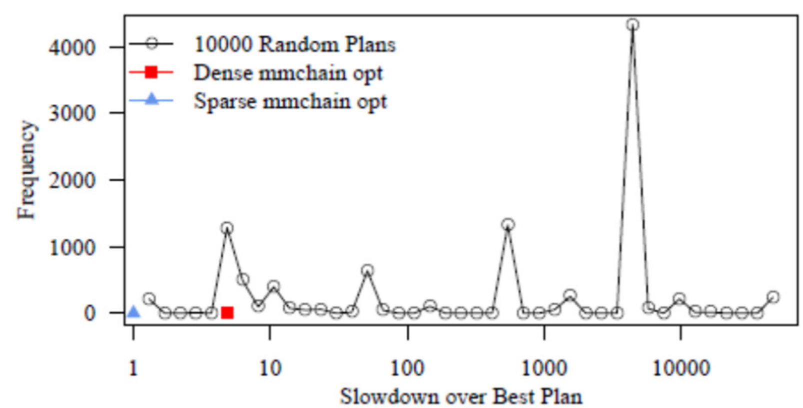


Sketch matrix E



  - Sketch propagation for optimal subchains (currently for all chains)
  - Modified cost computation via MNC sketches (**number FLOPs for sparse** instead of dense mm)

$$C_{i,j} = \min_{k\in[i,j-1]} \begin{pmatrix} C_{i,k} + C_{k+1,j} \\ + \mathbf{E_{i,k}.h^c E_{k+1,j}.h^r} \end{pmatrix}$$



Topic #2: Sparsity-Aware Optimization of Matrix Product Chains (incl DAGs)

Physical Rewrites and Optimizations

- **Distributed Caching**
  - Redundant compute vs. memory consumption and I/O
  - **#1 Cache intermediates w/ multiple refs** (Emma)
  - **#2 Cache initial read and read-only loop vars** (SystemML)

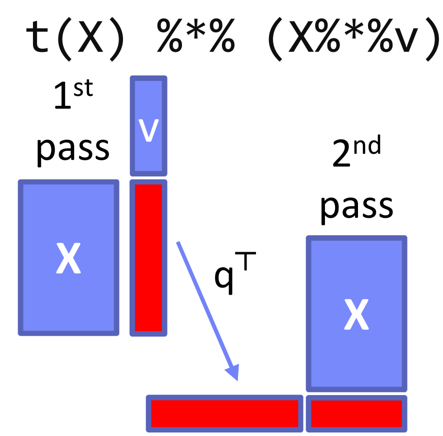
- **Partitioning**
  - Many frameworks exploit co-partitioning for efficient joins
  - **#1 Partitioning-exploiting operators** (SystemML, Emma, Samsara)
  - **#2 Inject partitioning to avoid shuffle per iteration** (SystemML)
  - **#3 Plan-specific data partitioning** (SystemML, Dmac, Kasen)

- **Other Data Flow Optimizations** (Emma)
  - **#1 Exists unnesting** (e.g., filter w/ broadcast → join)
  - **#2 Fold-group fusion** (e.g., groupByKey → reduceByKey)

- **Physical Operator Selection**

Physical Operator Selection

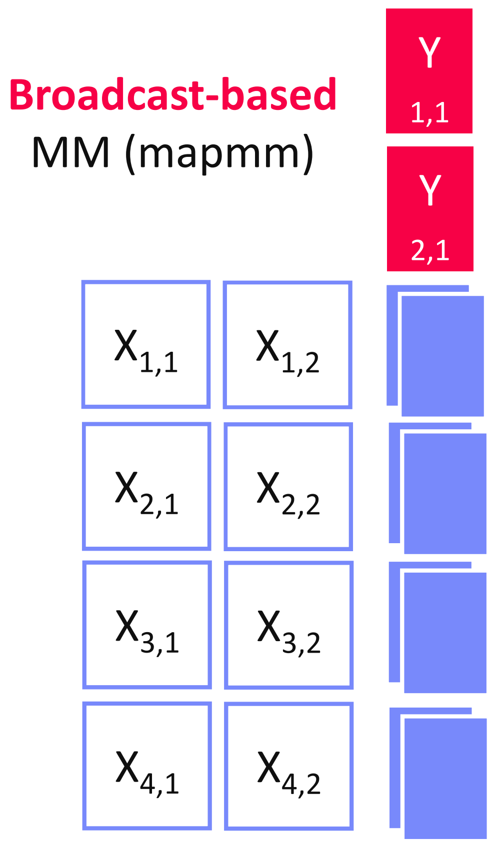
- **Common Selection Criteria**
  - **Data and cluster characteristics** (e.g., data size/shape, memory, parallelism)
  - **Matrix/operation properties** (e.g., diagonal/symmetric, sparse-safe ops)
  - **Data flow properties** (e.g., co-partitioning, co-location, data locality)
- **#0 Local Operators**
  - SystemML mm, tsmm, mmchain; Samsara/Mllib local
- **#1 Special Operators** (special patterns/sparsity)
  - SystemML **tsmm**, **mapmmchain**; Samsara AtA

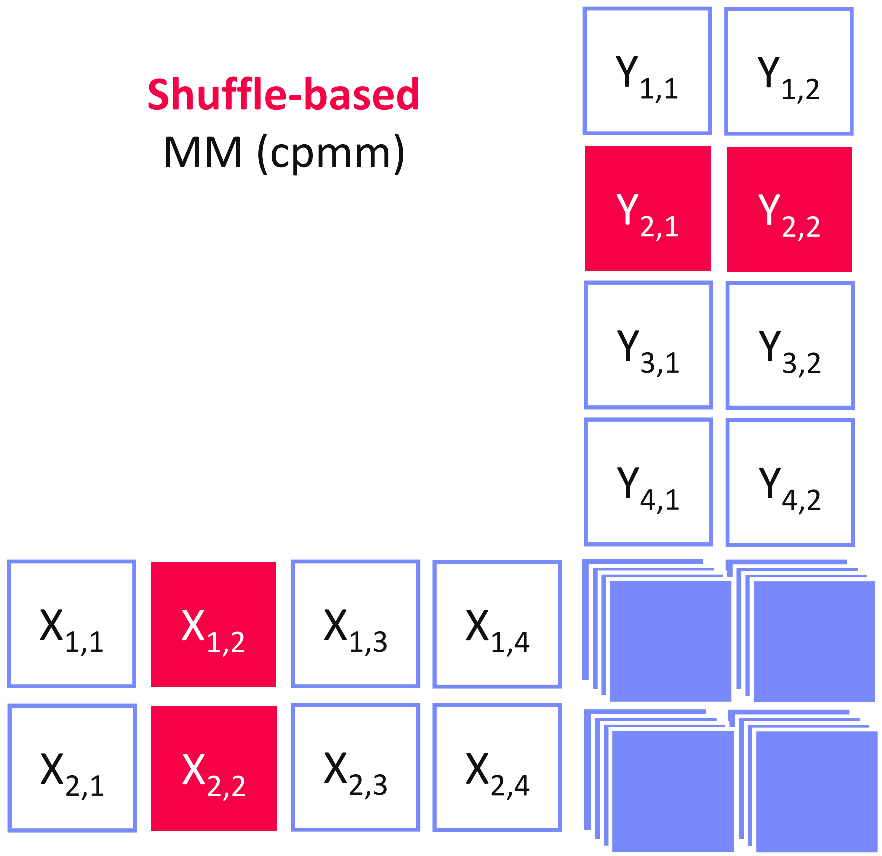


- **#2 Broadcast-Based Operators** (aka broadcast join)
  - SystemML **mapmm**, **mapmmchain**
- **#3 Co-Partitioning-Based Operators** (aka improved repartition join)
  - SystemML **zipmm**; Emma, Samsara OpAtB
- **#4 Shuffle-Based Operators** (aka repartition join)
  - SystemML **cpmm**, **rmm**; Samsara OpAB

Physical Operator Selection, cont.

- **Examples Distributed MM Operators**

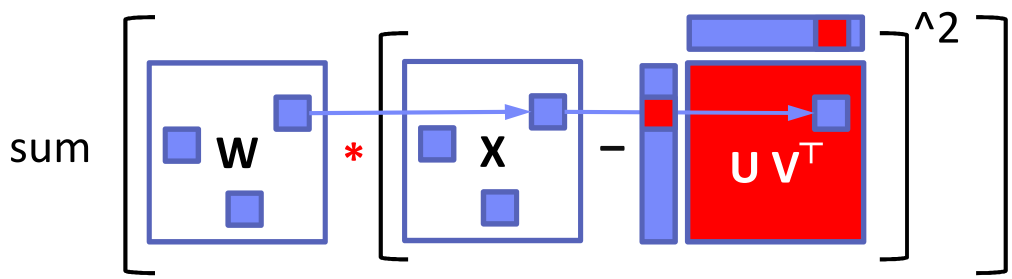




Sparsity-Exploiting Operators

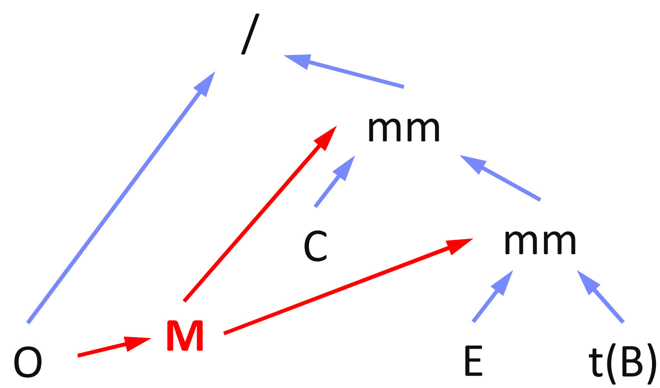
- **Goal:** Avoid dense intermediates and unnecessary computation
- **#1 Fused Physical Operators**
  - E.g., SystemML [PVLDB'16] wsloss, wcemm, wdivmm
  - Selective computation over non-zeros of "sparse driver"

`sum(W * (X – U %*% t(V))^2)`



- **#2 Masked Physical Operators**
  - E.g., Cumulon MaskMult [SIGMOD'13]
  - Create mask of "sparse driver"
  - Pass mask to single masked matrix multiply operator

`O / (C %*% E %*% t(B))`



Conclusions

- **Summary**
  - Basic compilation overview
  - Size inference and cost estimation (**foundation for optimization**)
  - **Rewrites and operator selection**
- **Impact of Size Inference and Costs**
  - Advanced optimization of linear algebra programs requires size inference for cost estimation and validity constraints
- **Ubiquitous Rewrite Opportunities**
  - Linear algebra programs have plenty of room for optimization
  - Potential for changed asymptotic behavior
- **Next Lectures**
  - **04 Operator Fusion and Runtime Adaptation** [Apr 05] (advanced compilation)

Backup: Programming/Analysis Projects

Example Projects (to be refined by Mar 29)

- **#1 Auto Differentiation**
  - Implement auto differentiation for deep neural networks
  - Integrate auto differentiation framework in compiler or runtime
- **#2 Sparsity-Aware Optimization of Matrix Product Chains**
  - Extend DP algorithm for DAGs and other operations
- **#3 Parameter Server Update Schemes**
  - New PS update schemes: e.g., stale-synchronous, Hogwild!
  - Language and local/distributed runtime extensions
- **#4 Extended I/O Framework for Other Formats**
  - Implement local readers/writers for NetCDF, HDF5, libsvm, and/or Arrow
- **#5 LLVM Code Generator**
  - Extend codegen framework by LLVM code generator
  - Native vector library, native operator skeletons, JNI bridge
- **#6 Reproduce Automated Label Generation** (analysis)

Example Projects, cont.

- **#7 Data Validation Scripts**
  - Implement recently proposed integrity constraints
  - Write DML scripts to check a set of constraints on given dataset
- **#8 Data Cleaning Primitives**
  - Implement scripts or physical operators to perform data imputation and data cleaning (find and remove/fix incorrect values)
- **#9 Data Preparation Primitives**
  - Extend `transform` functionality for distributed binning
  - Needs to work in combination w/ dummy coding, recoding, etc
- **#10 Common Subexpression Elimination & Constant Folding**
  - Exploit commutative common subexpressions
  - One-shot constant folding (avoid compile overhead)
- **#11 Repartition joins and binary ops without replication**
  - Improve repartition mm and binary ops by avoiding unnecessary replication