BUILDING FASTER DATA APPLICATIONS ON SPARK* CLUSTERS USING INTEL® DAAL

Zhang Zhang

Developer Products Division

Agenda

Overview - Apache Spark™

What is Intel® DAAL?

Intel® DAAL in the context of Spark

Speeding up Spark MLlib with Intel® DAAL

- Data conversion
- Programming model of Intel® DAAL distributed processing
- Example: PCA
- Example: Linear regression

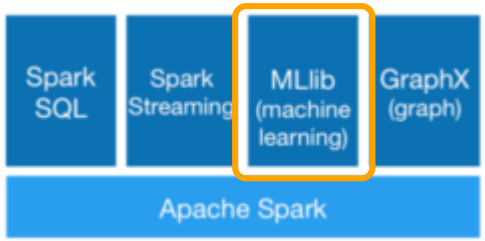
Performance

Future of Intel® DAAL

Overview of Apache Spark™

A fast and general engine for large-scale data processing*.

- Providing more operations than MapReduce
- Increasing developer productivity
- Running on Hadoop, Mesos, as a standalone, or in the cloud
- Fault-tolerant distributed data structures (RDD)
- A stack of powerful libraries



* Source: Apache Spark website (http://spark.apache.org)

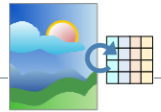
Intel® Data Analytics Acceleration Library
(Intel® DAAL)

An industry leading Intel® Architecture based data analytics acceleration library of fundamental algorithms covering all machine learning stages.

Pre-processing

(De-)Compression

Transformation



PCA
Statistical moments
Variance matrix
QR, SVD, Cholesky
Apriori

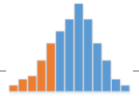
Analysis

Modeling

Linear regression
Naïve Bayes
SVM
Classifier boosting

Kmeans
EM GMM

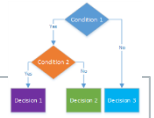
Validation



Collaborative filtering

Neural Networks

Decision Making

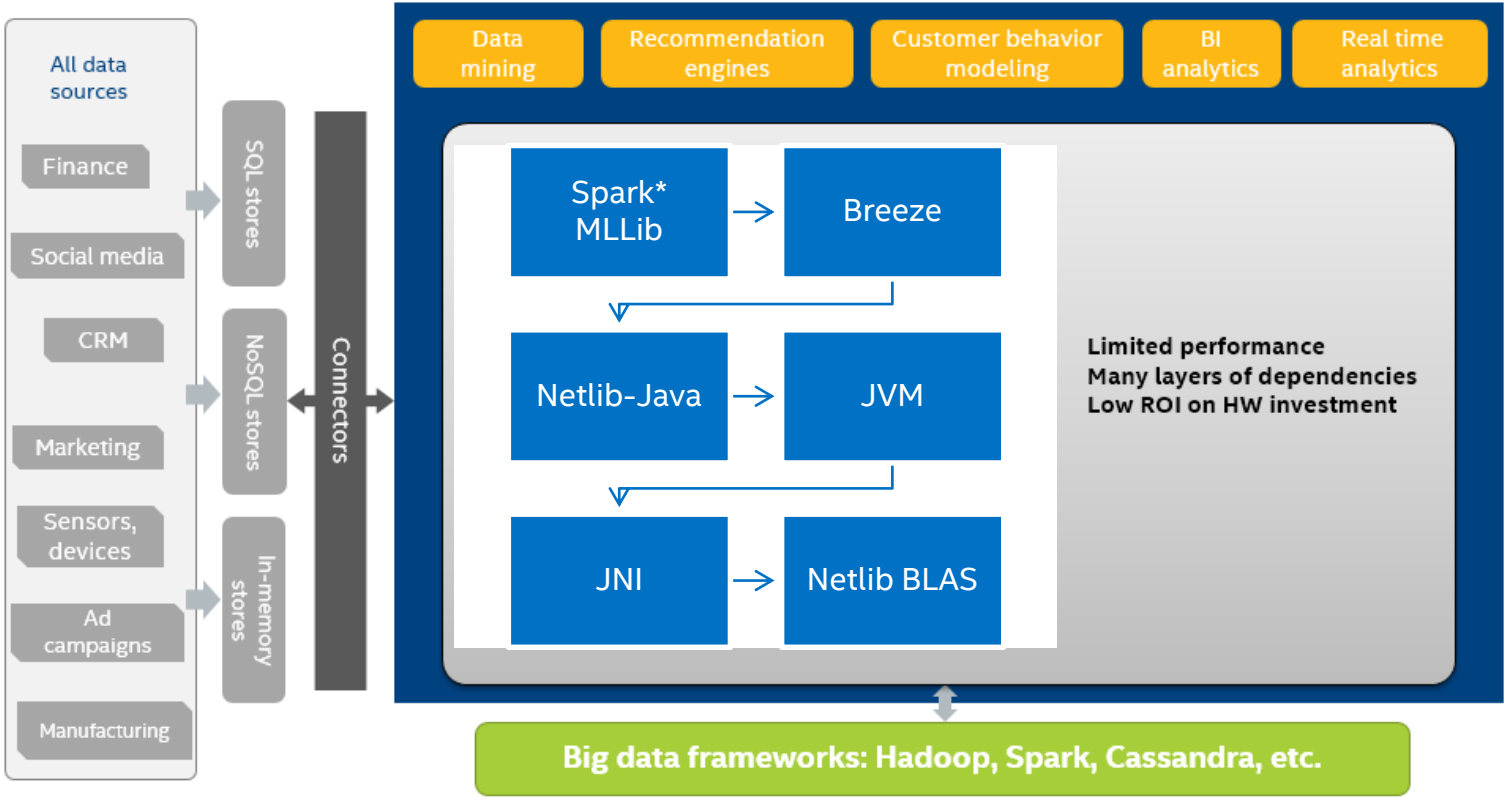


What's in the Package?

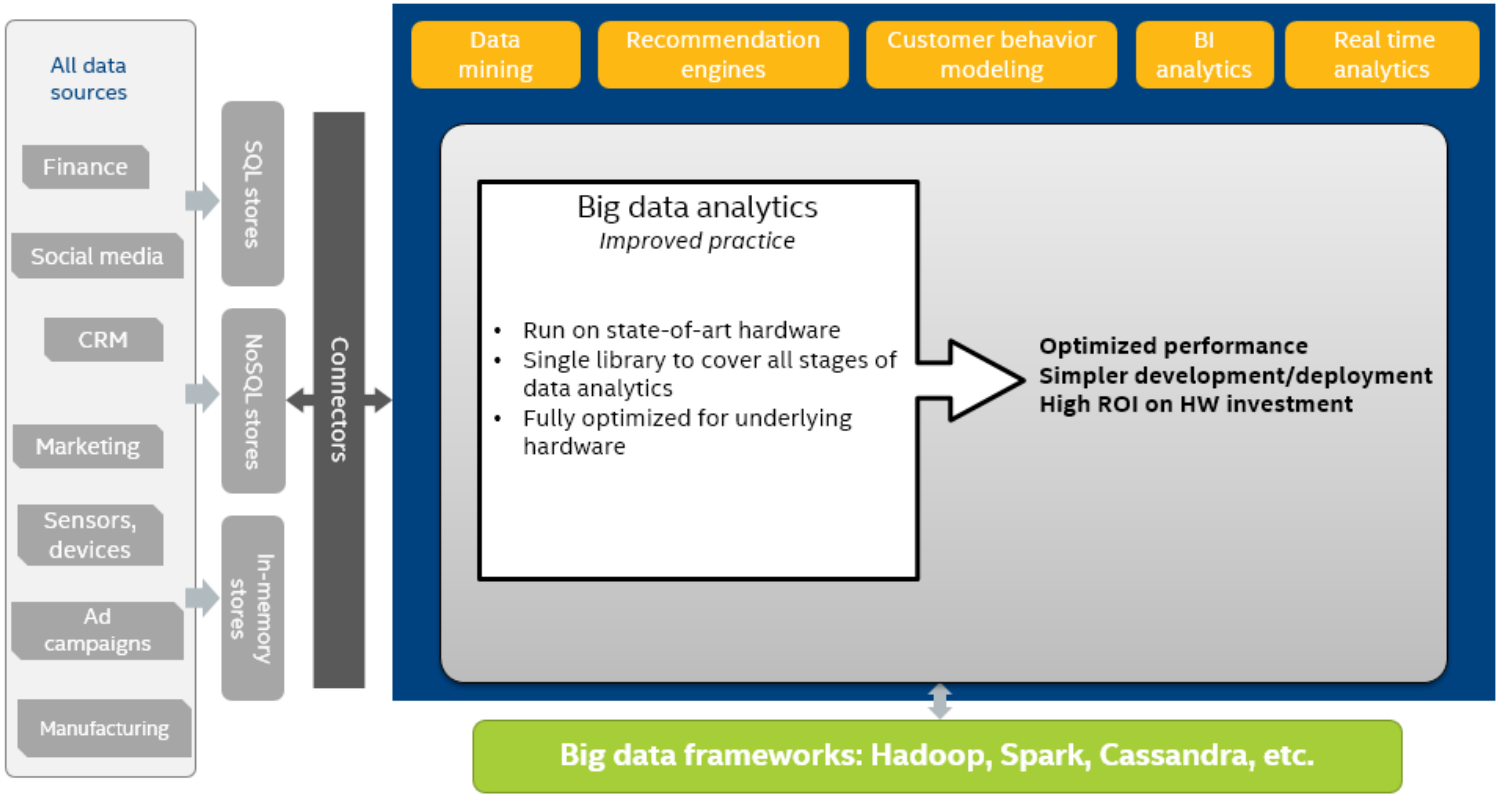
- Support IA-32 and Intel64 architectures
- Support Linux, Windows, and OS X*
- C++ and Java API. Python support is coming soon.
- Static and dynamic linking.
- A standalone library, and also bundled in Intel® Parallel Studio XE 2016.
- Also available as part of the **Intel Performance Libraries Community Edition** (free).
- An open source Intel DAAL will soon be available on Github.

Note: Bundled version is not available on OS* X.

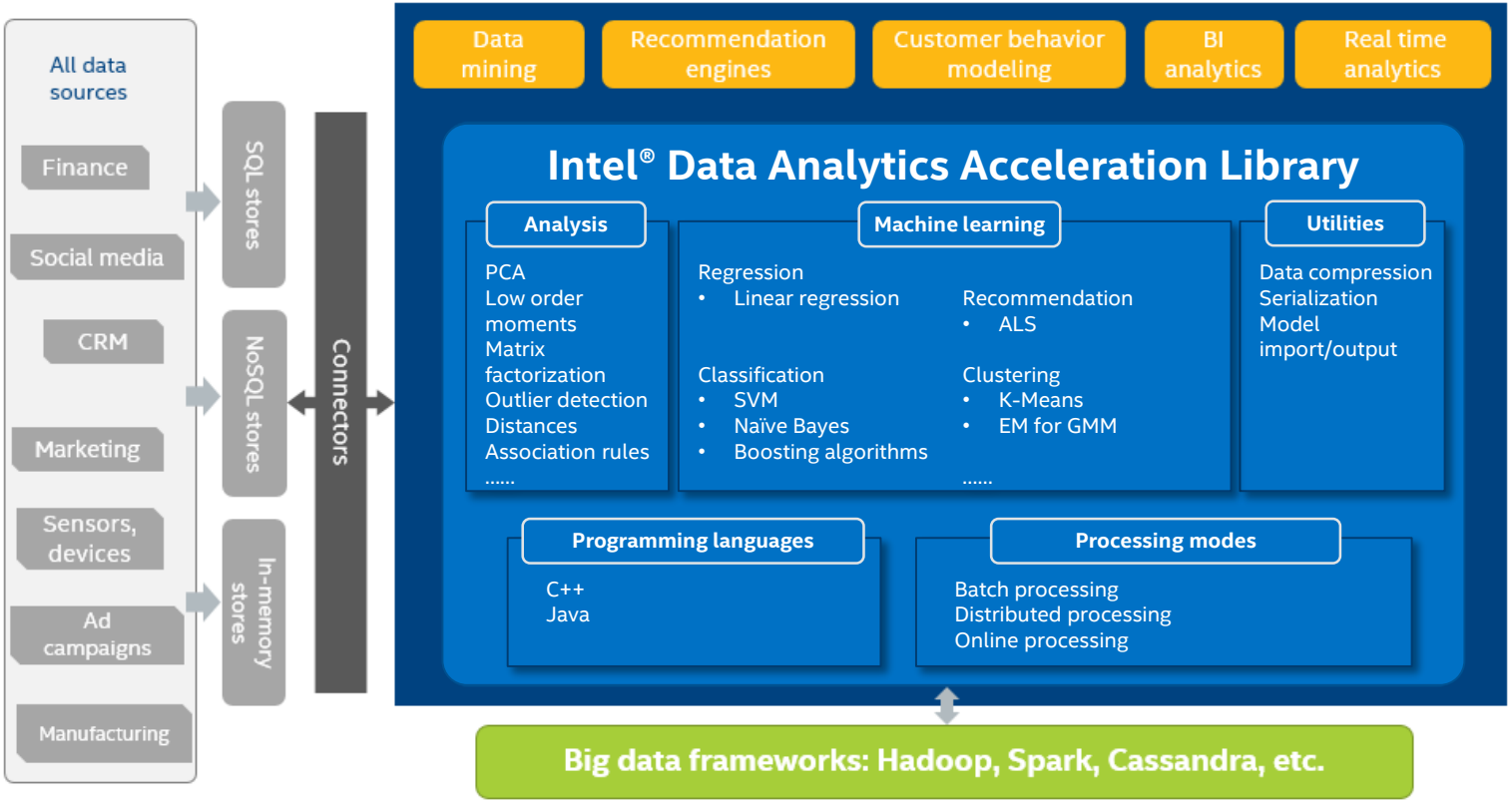
Where Intel DAAL Fits?



Where Intel DAAL Fits?

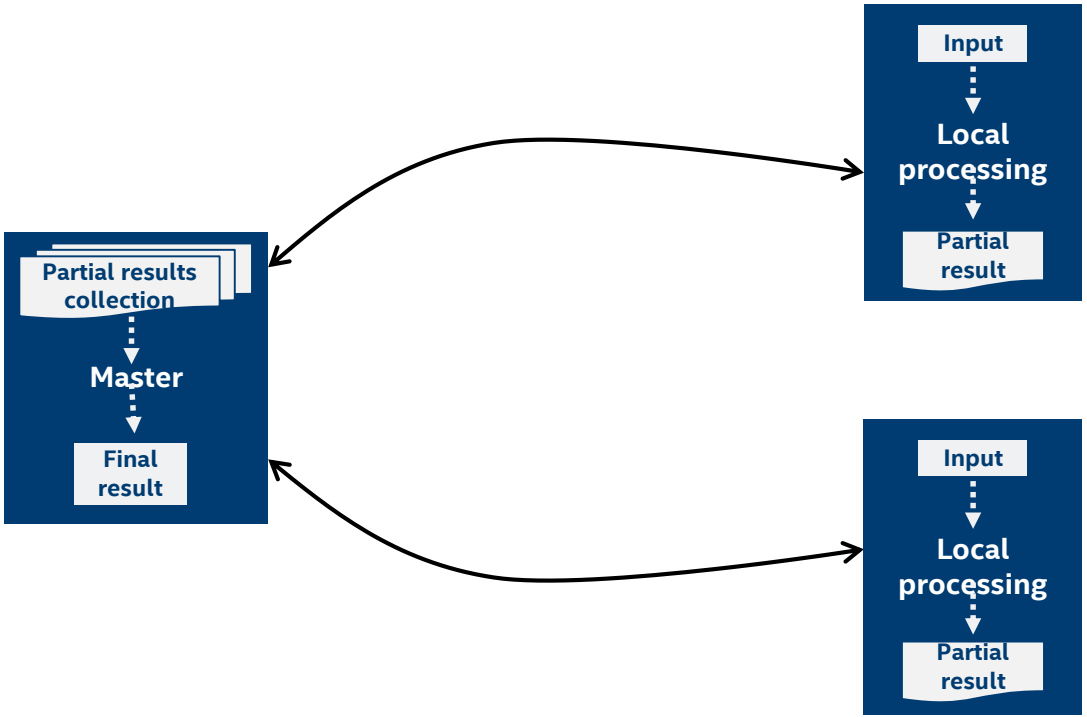


Where Intel DAAL Fits?

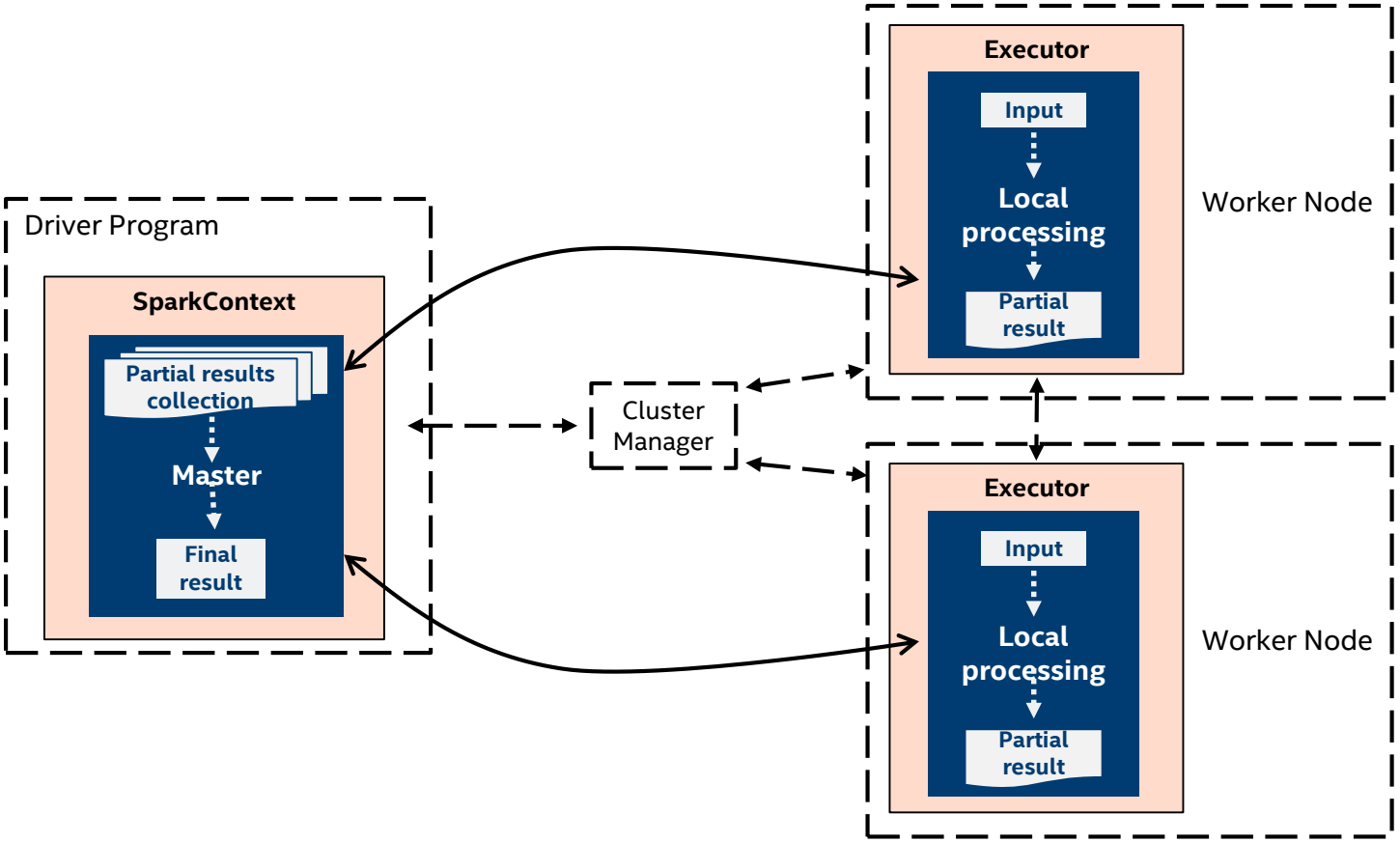


Intel® DAAL Distributed Processing

Conceptual Model



Intel DAAL in the Context of Spark



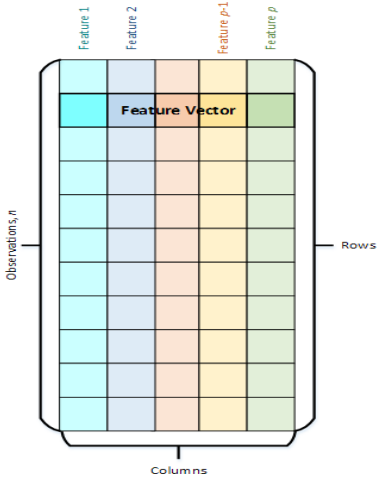
Intel DAAL Numeric Tables

Heterogeneous – AOS

- Observations are stored in contiguous memory buffers.

Heterogeneous – SOA

- Features are stored in contiguous memory buffers.



Homogeneous – Dense matrix

- 2D matrix: *n* rows (observations), *p* columns (features)

Homogeneous – Sparse matrix (CSR)

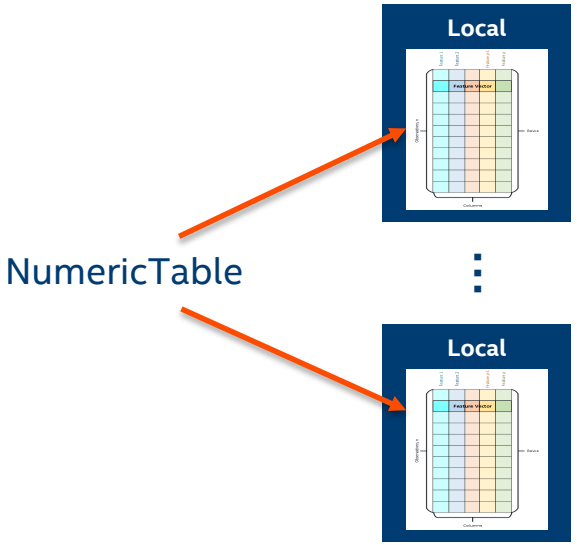
- Support both 0-based indexing and 1-based indexing.

m-by-n homogenous

$$\begin{bmatrix} x_{00} & x_{01} & \cdots & x_{0(n-1)} \\ x_{10} & x_{11} & \cdots & x_{1(n-1)} \\ \vdots & \vdots & \vdots & \vdots \\ x_{m0} & x_{m1} & \cdots & x_{m(n-1)} \end{bmatrix}$$

Numeric Tables and RDD

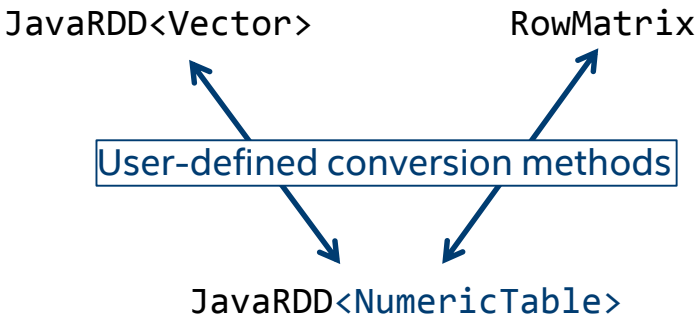
JavaRDD<NumericTable>




```
public class NumericTableWithIndex implements java.io.Serializable {

    private Tuple2<Long, NumericTable> tup;
    private long nRows;
    private long nCols;
```

Handle MLlib Distributed Data Structures



Data Conversion: RDD<Vector> to RDD<NumericTable>

```
JavaPairRDD<Vector, Long> vecrddWithIds = vecrdd.zipWithIndex();

JavaRDD<NumericTableWithIndex> jntrdd = vecrddWithIds.mapPartitions(
        new FlatMapFunction<Iterator<Tuple2<Vector, Long>>, NumericTableWithIndex>() {
            public List<NumericTableWithIndex> call(Iterator<Tuple2<Vector, Long>> it) {
                DaalContext context = new DaalContext();
                ArrayList<NumericTableWithIndex> tables = new ArrayList<NumericTableWithIndex>();
                int cursize = 0;
                int nrows = 0;
                double[] data = new double[0];
                while (it.hasNext()) {
                    Tuple2<Vector, Long> tup = it.next();
                    double[] row = tup._1().toArray();
                    data = ArrayUtils.addAll(data, row);
                    cursize += row.length;
                    nrows++;
                    if (nrows == maxRowsPerTable || !it.hasNext()) {
                        NumericTableWithIndex part = new NumericTableWithIndex(
                                tup._2() - nrows + 1,
                                new HomogenNumericTable(context, data, cursize/nrows, nrows));
                        tables.add(part);
                        cursize = 0;
                        nrows = 0;
                    }
                }
                return tables;
            }
        }, true);

return new DistributedNumericTable(jntrdd, vecrdd.count(), ncols);
```

Data Conversion: RDD<NumericTable> to RDD<Vector>

```
return distNT.numTables.flatMap(
        new FlatMapFunction<NumericTableWithIndex, Vector>() {
            public List<Vector> call(NumericTableWithIndex nt) {
                DaalContext context = new DaalContext();
                NumericTable table = nt.getTable(context);
                double[] data = (table instanceof HomogenNumericTable) ?
                                ((HomogenNumericTable) table).getDoubleArray() : null;
                if (data == null) {
                    throw new IllegalArgumentException("Invalid NumericTable type");
                }
                long begin = 0;
                long end = nt.numOfCols();
                ArrayList<Vector> veclist = new ArrayList<Vector>();
                while (begin < data.length) {
                    double[] row = ArrayUtils.subarray(data, (int)begin, (int)end);
                    DenseVector dv = new DenseVector(row);
                    veclist.add(dv);
                    begin = end;
                    end += nt.numOfCols();
                }
                context.dispose();
                return veclist;
            }
        });
```

Programming Model of Distributed Processing
Local Processing

Local processing algorithms abstractions:

- Package: `com.intel.daal.algorithms`
  - `pca.DistributedStep1Local`
  - `linear_regression.Training.TrainingDistributedStep1Local`
  - ……

Partial results abstractions:

- `pca.PartialResult`
- `linear_regression.Training.PartialResult`
- ……

```
For each table in JavaRDD<NumericTable>:

    // Local processing algorithm and params
    DistributedStep1Local alg =
        new DistributedStep1Local(……);

    // Set input
    alg.input.set(Input.data, table);

    // Compute and access partial result
    PartialResult ret = alg.compute();
        …
```

Programming Model of Distributed Processing
Master Side

Master side algorithms abstractions:

- Package: `com.intel.daal.algorithms`
  - `pca.DistributedStep2Master`
  - `linear_regression.Training.TrainingDistributedStep2Master`
  - ……

Final results abstractions:

- `pca.Result`
- `linear_regression.Training.TrainingResult`
- ……

```
// Collect partial results from all slaves
List<PartialResult> partsList = parts.collect();

// Master side processing algorithms and params
DistributedStep2Master alg =
                    new DistributedStep2Master(……);

// Set master side processing input
for (PartialResult val : partsList) {
    ……
    alg.input.add(MasterInputId.partialResults, val);
}

// Compute
alg.compute();

// Get final result
Result result = alg.finalizeCompute();
```

Example: Intel DAAL PCA on Spark

Intel DAAL provides two computation methods:

- Correlation method (*default)*
- SVD method

Input in the form of NumericTables:

- Non-normalized data, or
- Normalized data ($\mu = 0$, $\sigma = 1$), or
- Correlation matrix

Output:

- Scores (A *1xp NumericTable* with eigenvalues, largest to smallest)
- Loadings (A *pxp NumericTable* with corresponding eigenvectors)

PCA - Slave Side

```
final Broadcast<Tuple2<Class<? extends Number>, PCAMethod>> configBcast = sc.broadcast(config);
// Local processing on all slaves
JavaRDD<PartialResult> partsrdd = nTables.map(
        new Function<NumericTableWithIndex, PartialResult>() {
            public PartialResult call(NumericTableWithIndex table) {
                DaalContext context = new DaalContext();
                // Create algorithm to calculate PCA decomposition using Correlation method on local nodes
                DistributedStep1Local pcaLocal = new DistributedStep1Local(
                        context,
                        configBcast.value()._1(),
                        configBcast.value()._2().getMethod());

                // Set input data on local node
                pcaLocal.input.set(InputId.data, table.getTable(context));

                // Compute PCA on local node
                PartialResult pres = pcaLocal.compute();
                pres.pack();

                context.dispose();
                return pres;
            }
        }).cache();
```

PCA – Master Side

```
// Finalize on master
List<PartialResult> partscollection = partsrdd.collect();
DistributedStep2Master pcaMaster = new DistributedStep2Master(dc, config._1(), config._2().getMethod());
for (PartialResult value : partscollection) {
    value.unpack(dc);
    pcaMaster.input.add(MasterInputId.partialResults, value);
}
pcaMaster.compute();
Result daalresult = pcaMaster.finalizeCompute();

return new PCAResult(daalresult.get(ResultId.eigenValues),
                     daalresult.get(ResultId.eigenVectors));
```

Example: Intel DAAL Linear Regression on Spark

Training – distributed processing

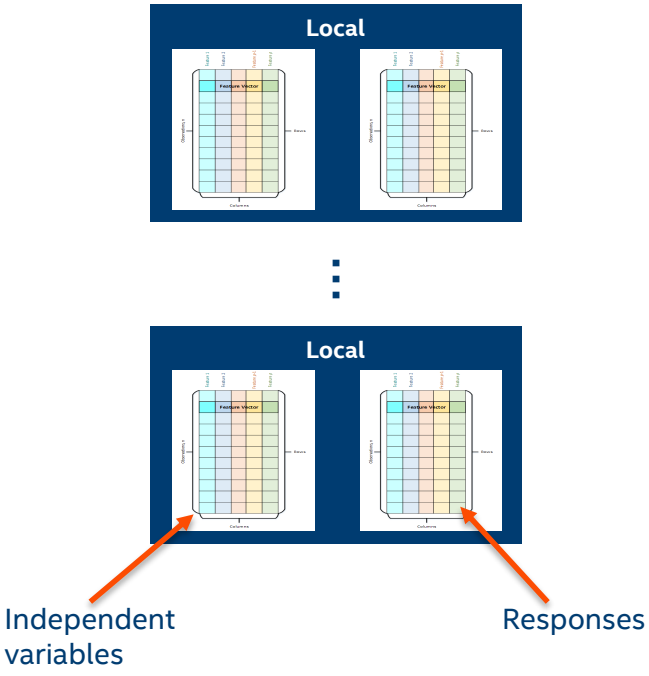
- Computation methods:
  - Normal equation method
  - QR method
- Input:
  - An *nxp NumericTable* of independent variables
  - An *nxk NumericTable* of corresponding known responses
- Output:
  - A Model object (the intercept, coefficients)

Prediction – batch processing, on master side only

- Input:
  - A Model
  - An *mxp NumericTable* of unseen data
- Output:
  - An *mxk NumericTable* of predicted responses

Training Data Representation

```
JavaPairRDD<NumericTable, NumericTable>
```



Linear Regression Model Training – Slave Side

```
final Broadcast<Tuple2<Class<? extends Number>, Method>> config = sc.broadcast(trConfig);
// Local processing on all slaves
JavaRDD<PartialResult> partsrdd = dataWithLabels.map(
        new Function<Tuple2<NumericTable, NumericTable>, PartialResult>() {
            public PartialResult call(Tuple2<NumericTable, NumericTable> tup) {
                DaalContext context = new DaalContext();

                // Create algorithm to train a model
                TrainingDistributedStep1Local training = new TrainingDistributedStep1Local(
                        context, config.value()._1(), config.value()._2().getMethod());
                // Set input data on local node
                tup._1().unpack(context);
                tup._2().unpack(context);
                training.input.set(TrainingInputId.data, tup._1());
                training.input.set(TrainingInputId.dependentVariable, tup._2());
                // Compute on local node
                PartialResult pres = training.compute();
                pres.pack();

                context.dispose();
                return pres;
            }
        }).cache();
```

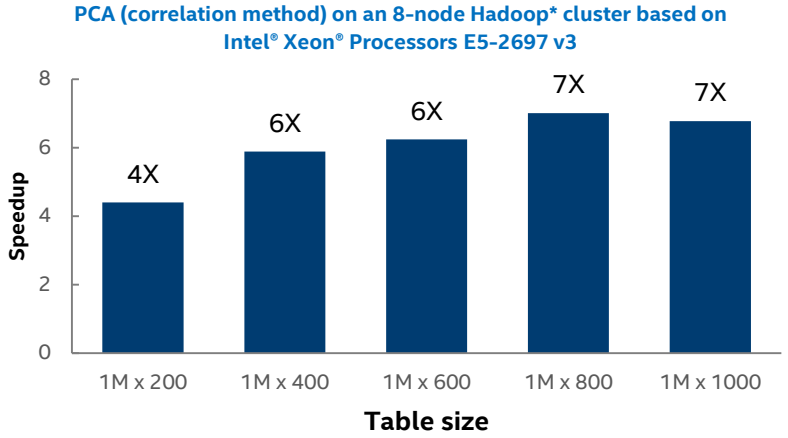
Linear Regression Model Training – Master Side

```
// Finalizing on master
List<PartialResult> partscollection = partsrdd.collect();
TrainingDistributedStep2Master master =
        new TrainingDistributedStep2Master(dc, trConfig._1(), trConfig._2().getMethod());
for (PartialResult value : partscollection) {
    value.unpack(dc);
    master.input.add(MasterInputId.partialModels, value);
}
master.compute();
TrainingResult result = master.finalizeCompute();
return result.get(TrainingResultId.model);
```

Linear Regression Prediction

```
// Linear Regression prediction only works with batch mode.
// Prediction algorithm
PredictionBatch predict = new PredictionBatch(dc, predictFpType, predictMethod);
// Set input
predict.input.set(PredictionInputId.data, testData);
// Set model
predict.input.set(PredictionInputId.model, model);
return predict.compute();
```

PCA Performance Boosts Using Intel® DAAL vs. Spark* MLLib on Intel® Architectures



Configuration Info - Versions: Intel® Data Analytics Acceleration Library 2016, CDH v5.3.1, Apache Spark* v1.2.0; Hardware: Intel® Xeon® Processor E5-2699 v3, 2 Eighteen-core CPUs (45MB LLC, 2.3GHz), 128GB of RAM per node; Operating System: CentOS 6.6 x86_64.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. * Other brands and names are the property of their respective owners. Benchmark Source: Intel Corporation

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804 .

DAAL 2017 Key New Features

Neural Networks

Python API (a.k.a. PyDAAL)

Open source project on Github

Join the Beta program today
https://softwareproductsurvey.intel.com/f/150587/1103/

Intel® DAAL Neural Networks Components

Layers

Common layers	Activation	Normalization
Convolutional	Logistic	Z-score
Pooling (max, average, stochastic, spatial)	Hyperbolic tangent	Batch
Fully connected	ReLU, pReLU, soft ReLU	Local response
Locally connected	Softmax	Local contrast
Dropout	Abs	

Optimization solvers

Supported objective functions
MSE (mean squared errors)
Supported solvers
SGD
Mini-batch gradient descent
Stochastic LBFGS
Adagrad

Python API (a.k.a. PyDAAL)

Stick closely with DAAL's overall design

- Object-oriented, namespace hierarchy, plug&play

Seamless interfacing with NumPy

Anaconda package

- http://anaconda.org/intel/

```
...

# Create a Numpy array as our input
a = np.array([[1,2,4],
              [2,1,23],
              [4,23,1]])

# create a DAAL Matrix using our numpy array
m = daal.Matrix(a)

# Create algorithm objects for cholesky decomposition
computing using default method
algorithm = cholesky.Batch()

# Set input arguments of the algorithm
algorithm.input.set(cholesky.data, m)

# Compute Cholesky decomposition
res = algorithm.compute()

# Get computed Cholesky decomposition
tbl = res.get(choleskyFactor)

# get and print the numpy array
print tbl.getArray()
```

Open Source Project

Co-exists with the proprietary version

Apache 2.0 license

Lives on github.com

Resources

Product Links

- Intel® Data Analytics Acceleration Library
  - https://software.intel.com/en-us/intel-daal
- User forum
  - https://software.intel.com/en-us/forums/intel-data-analytics-acceleration-library
- Community licensing program
  - https://software.intel.com/sites/campaigns/nest

Code Modernization Links

- Modern Code Developer Community
  - software.intel.com/modern-code
- Intel Code Modernization Enablement Program
  - software.intel.com/code-modernization-enablement
- Intel Parallel Computing Centers
  - software.intel.com/ipcc
- Technical Webinar Series Registration
  - http://bit.ly/spring16-tech-webinars
- Intel Parallel Universe Magazine
  - software.intel.com/intel-parallel-universe-magazine

Call to Action

Download the code samples and try it out on your Spark cluster

- https://software.intel.com/en-us/articles/using-intel-data-analytics-acceleration-library-on-apache-spark

Join the 2017 Beta program

- Intel DAAL is part of Intel Parallel Studio XE 2017 Beta
- Registration link
  - https://softwareproductsurvey.intel.com/f/150587/1103/

Legal Disclaimer & Optimization Notice

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

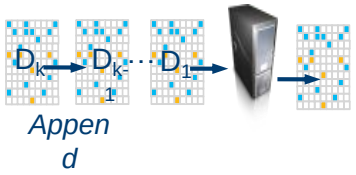
Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

BACKUP SLIDES

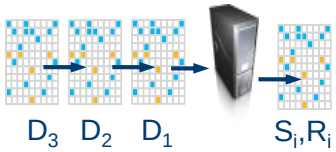
Intel DAAL Processing modes

Batch Processing



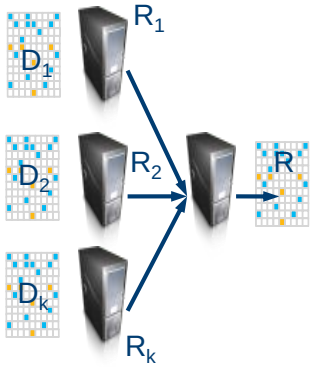
$R = F(D_1,\ldots,D_k)$

Online Processing



$S_{i+1} = T(S_i,D_i)$
$R_{i+1} = F(S_{i+1})$

Distributed Processing



$R = F(R_1,\ldots,R_k)$

	Algorithms	Batch	Distributed	Online
Descriptive statistics	Low order moments	√	√	√
	Quantiles	√		
Statistical relationships	Correlation / Variance-Covariance	√	√	√
	(Cosine, Correlation) distance matrices	√		
Matrix decomposition	SVD	√	√	√
	Cholesky decomposition	√		
	QR decomposition	√	√	√
Regression	Linear regression	√	√	√
Classification	Naïve Bayes	√	√	√
	SVM (two-class and multi-class)	√		
	Boosting	√		
Unsupervised learning	Association rules mining	√		
	Anomaly detection	√		
	PCA	√	√	√
	KMeans	√	√	
	EM for GMM	√		
Recommender systems	ALS	√	√	
Deep learning	Neural networks	√	√	


intel®
experience
what's inside™