

Amazon Product Rating Predictions at Scale

Yuya Jeremy Ong & Yiyue Zou

DS 410 - Data Analytics at Scale

Recommender Systems Powers Modern E-Commerce

Retailers must provide good recommendations to bridge the gap between customers to products.



Objectives

1. Implement CF-based model which predicts the user's rating of a product based on various features provided by the dataset.
2. Systematically execute the various model configurations across different optimization parameters and evaluate their performance using various profiling tools provided by Spark.
3. Conduct a systematic investigation of performance bottlenecks and search for potential areas of optimizations and improvements to original implementations.
4. Perform a trade-off analysis of the various methods based on quantitative and qualitative metrics obtained from our experiments.

Model Requirements and Objectives

Our model should be able to perform the following:

1. Ideally we would like to train our model on a per-daily basis to ensure that our system's rating/recommendations are up-to-date.
2. Generate a “good-enough” or “reasonable” prediction.
3. Devise scalable architecture which can be scaled by hardware-infrastructure defined optimizations.*

*The primary focus of our project will be based on the third objective.

Amazon Product Review Dataset

- Contains **148.2 mil.** product reviews and metadata information from Amazon spanning between May 1996 to July 2014.
- Dataset includes Reviews, Product Metadata and Links.
- All PII related information were removed and replaced with randomly generated IDs.
- Compiled by Professor Julian McAuley from UCSD.

- For our implementation, we have only utilized a smaller subset of the dataset due to feasibility and narrower scope.
- We further narrow the scope of the recommender system to mitigate issues with sparsity (*more on this later*).
- We have utilized the **Books** category which contains the following number of records:
 - Total Product Reviews: 8,898,041 Records
 - Total Products: 2,370,585 Records

Dataset Schema

Books Review Dataset

```
root
|-- asin: string (nullable = true)
|-- helpful: array (nullable = true)
|   |-- element: long (containsNull = true)
|-- overall: double (nullable = true)
|-- reviewText: string (nullable = true)
|-- reviewTime: string (nullable = true)
|-- reviewerID: string (nullable = true)
|-- reviewerName: string (nullable = true)
|-- summary: string (nullable = true)
|-- unixReviewTime: long (nullable = true)
```

Books Metadata Dataset

```
root
|-- _corrupt_record: string (nullable = true)
|-- asin: string (nullable = true)
|-- brand: string (nullable = true)
|-- categories: array (nullable = true)
|   |-- element: array (containsNull = true)
|       |-- element: string (containsNull = true)
|-- description: string (nullable = true)
|-- imUrl: string (nullable = true)
|-- price: double (nullable = true)
|-- related: struct (nullable = true)
|-- title: string (nullable = true)
```

Exploratory Summary Statistics

User-Review Statistics

- Count: 603,668
- Mean: 14.7399
- Std Dev: 51.7764
- Min: 5
- Max: 23,222

Product-Review Statistics

- Count: 3,679,982
- Mean: 24.1806
- Std Dev: 66.3029
- Min: 5
- Max: 7440

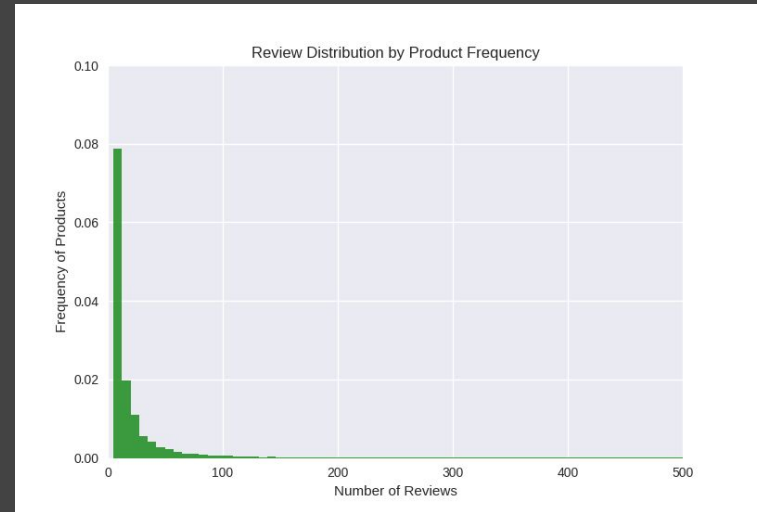
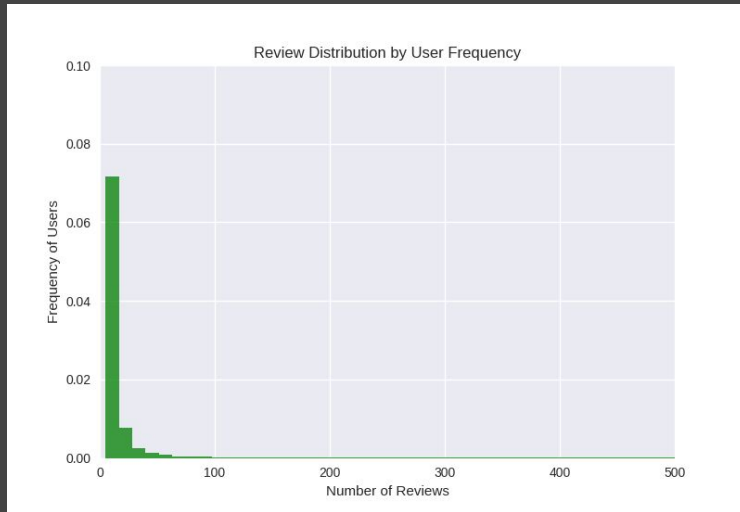
Helpfulness Statistics

- Count: 4,756,837
- Mean: 0.7349
- Std Dev: 0.3427
- Min: 0.0
- Max: 1.0



Review Distributions

Data has **strong positive skew**, which implies that our **data suffers from sparsity**.
This is a often common problem with dealing with large scale datasets.



Note: Graph windows have been cropped up to 500 since there isn't much anything interesting at the tail ends.

Handling Data Sparsity

To mitigate the issue of sparsity, we applied a cut-off threshold to filter any samples with fewer than a certain number of reviews.

Doing so would help in the following ways:

1. Scaling down our dataset to an even manageable size.
2. Increasing confidence and reliability of models.

After parsing the dataset, we have applied a conditional filter function which removes any review (whether its against user or product) that has less than the corresponding $\text{ceil}(\text{mean}) - 1$.

User-Review: $\text{ceil}(14.7399) - 1 = 14$

Product-Review: $\text{ceil}(24.8106) - 1 = 24$

Preprocessing Results

User-Review Statistics

- Count: 134,354 (47% Decrease)
- Mean: 40.9745 (178% Increase)
- Std Dev: 105.5534 (104% Increase)
- Min: 15 (200% Increase)
- Max: 23,222 (Not Affected)

Product-Review Statistics

- Count: 77,605 (98% Decrease)
- Mean: 77.3837 (220% Increase)
- Std Dev: 131.0095 (98% Increase)
- Min: 25 (400% Increase)
- Max: 7440 (Not Affected)

Result: Reduced our feature matrix from **2,221,487,373,976** to **10,426,542,170** records
That's an 82.58% decrease

Problem Formulation

Objective: Given a user's past set of review records, predict a rating for a product for which a user has not purchased yet.

Notation

u	=	User
p	=	Product
$r(u_i, p_j)$	=	Function which denotes the presence of a rating. (<i>1 if present, 0 if not</i>)
R	=	User-to-Product Matrix, where $R \in \mathbb{R}^{m \times n}$ (for m users and n items)
$R_{i,j}$	=	Denotes a rating from user i to product j

Collaborative Filtering

Collaborative Filtering is a algorithm typically implemented for generating recommendations.

This algorithm is generally content agnostic and relies primarily on a source of collective interactions between the users and products to base its predictions on.

CF relies on the premise that **users who have similar tastes or interests** are most likely to agree or share the same preferences for a given item.

Spark provides an implementation of Collaborative Filtering based on Hu et. al, 2008.

Social Information filtering exploits similarities between the tastes of different users to recommend (or advise against) items. It relies on the fact that people's tastes are not randomly distributed: there are general trends and patterns within the taste of a person and as well as between groups of people. Social Information filtering automates a process of "word-of-mouth" recommendations. A significant difference is that instead of having to ask a couple friends about a few items, a social information filtering system can consider thousands of other people, and consider thousands of different items, all happening autonomously and automatically (Shardanand and Maes, 1995).

Alternating Least Squares Model

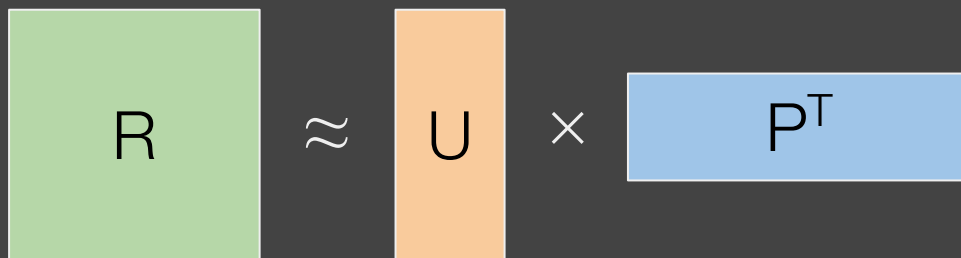
MLlib uses a model-based CF approach known as **Alternating Least Squares (ALS)**, which utilizes a Matrix Factorization algorithm proposed by Koren et. al, 2009.

Matrix Factorization allows to learn latent features, or in this case, hidden interactions between users and products.

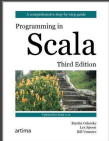

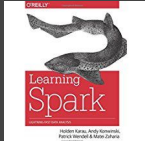


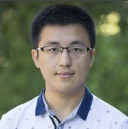


This process in a nutshell tries to “factorize”, or finds two matrices such that when you multiply the two together you would obtain the original matrix.

Concretely this would entail finding two such matrices, U and P which approximates the original user-to-item rating matrix. To obtain such rating of a product, we would take the dot product between the corresponding user and product vector.

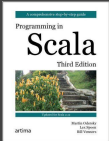

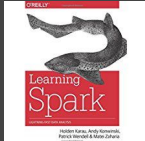


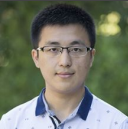


$$R \approx UP^T$$



Concrete Example of User-Based CF

				
	5		4	1
	4			1
	2	1		5
		2	4	5

Concrete Example of User-Based CF

				
	5		4	1
	4		4.5	1
	2	1		5
		2	4	5

Computing the Latent Factors

To compute the values for U and P using Matrix Factorization, we use an iterative algorithm which attempts to minimize the following cost function (and regularization term):

$$\min_{U, P} J = \boxed{\|R - UP^T\|_2} + \boxed{\lambda (\|U\|_2 + \|P\|_2)}$$

Error Term

Regularization Term

More explicitly:

$$\min_{X, Y} \boxed{\sum_{r(u, p)} (r_{up} - x_u^T y_p)^2} + \boxed{\lambda (\sum_u \|x_u\|^2 + \sum_p \|y_p\|^2)}$$

Through this process, we choose a parameter k , a fixed integer, which denotes the parameter vector's dimension that is used to summarize the user and product variables.

Using the error defined above, we learn the parameters of each of the k -dimensional feature vectors

Single Machine Implementation

The following pseudo-code describes the process for a single machine implementation:

```
initialize U, P  
for 1  $\rightarrow$  num_epoch (or until convergence):
```

```
    // Update User Parameters  
    for j = 1  $\rightarrow$  n:  
        // Update  $k$  weights for users (keeping product weights fixed)  
    end for
```

```
    // Update Product Parameters  
    for j = 1  $\rightarrow$  m:  
        // Update  $k$  weights for products (keeping user weights fixed)  
    end for
```

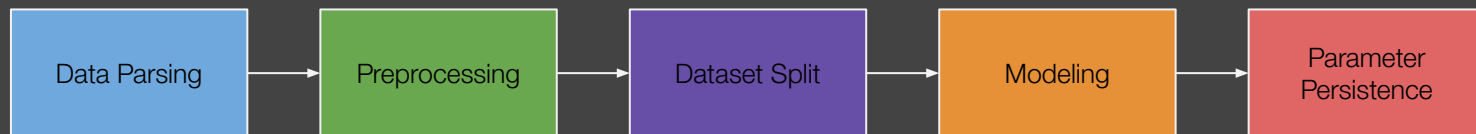
```
end for
```

Possible areas for
parallelization!

RDD Lineage Chain Pipeline

The following slides will discuss the RDD data flow described by a corresponding DAG diagram.

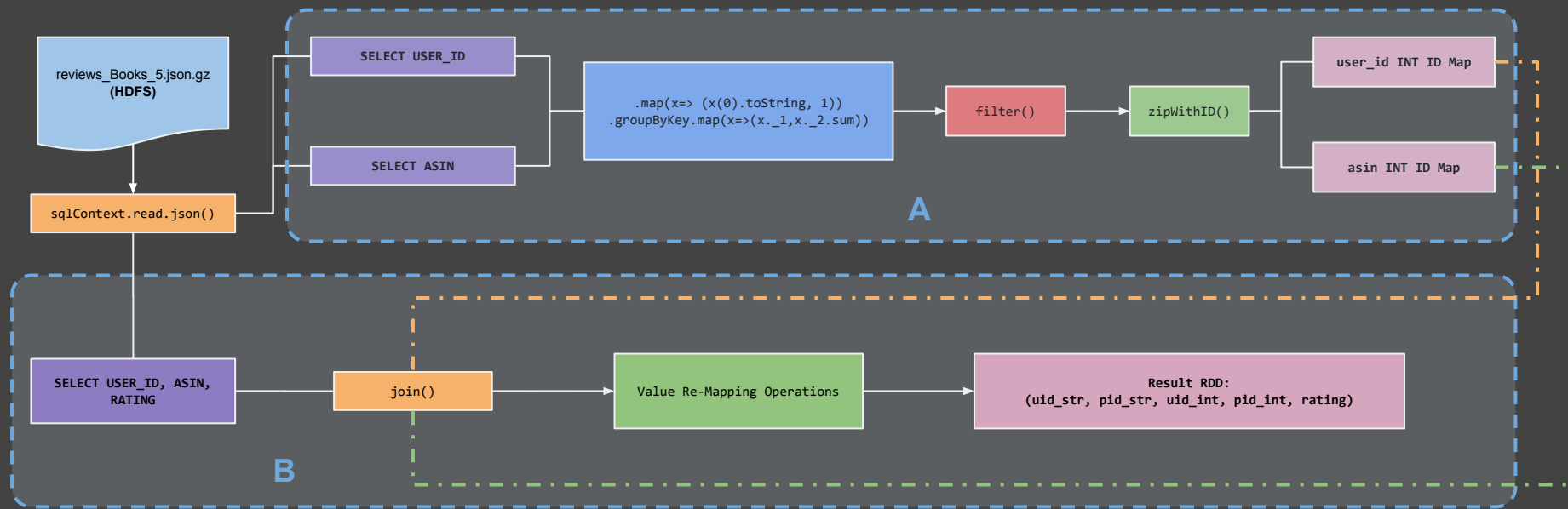
We first present the high-level processes involved in the model development pipeline.



Data Parsing and Preprocessing

The dataset parsing requires a two step process:

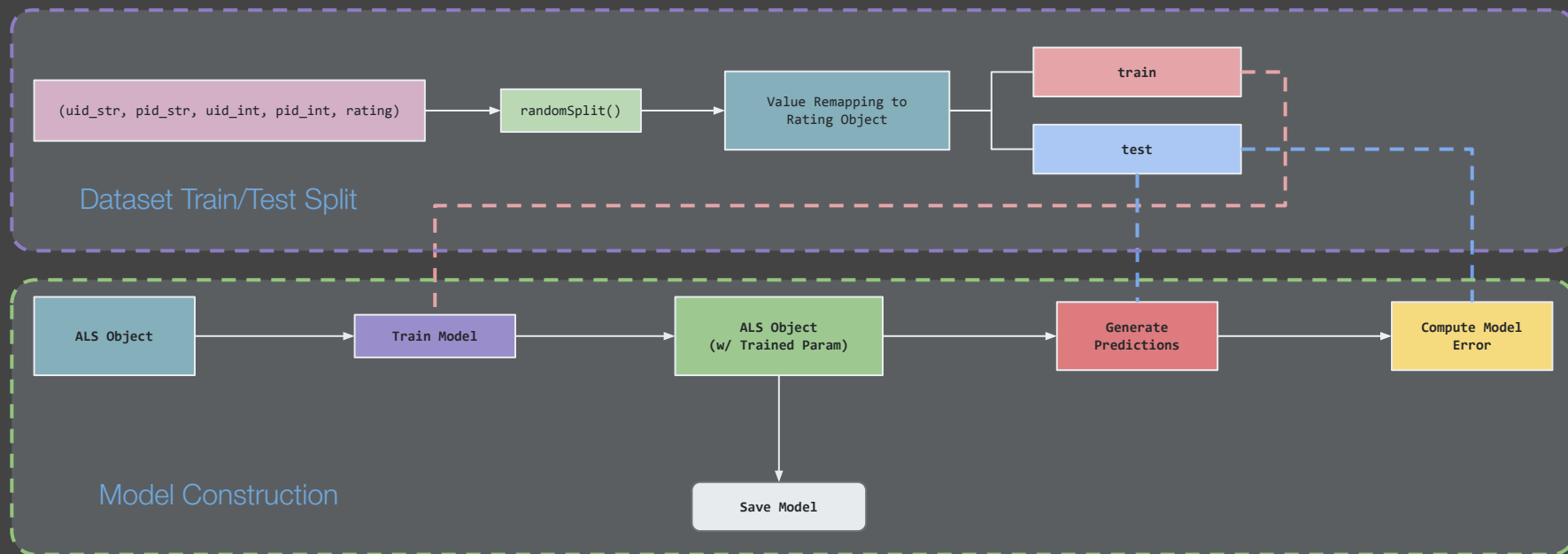
- A. Identifying key user_id and product IDs which satisfy our frequency conditions.
- B. Converting string identifiers to integers for use in MLlib.



Dataset Split and Model Construction

Prior to generating our models, we will need to split our dataset into training and test to evaluate our model accuracy/error.

The following RDD lineage chain describes the process behind the split and modeling process.



ALS Parameters

For our experiment, we have kept the model parameters constant throughout all of our performance validation trials.

Below are the parameters we have utilized in our experiments:

Rank (k): 10

Number of Iterations: 20

Learning Rate: 0.01

Model Results

Evaluation Metric: Root Mean Squared Error

$$\text{RMSE} = \sqrt{\sum \frac{(y_{pred} - y_{ref})^2}{N}}$$

Average Mean Squared Error:

We evaluate the performance of our model based on the conditional filtering introduced previously.

Without Filtering: 3.3724

With Data Filtering: 1.8256

Spark-Submit Parameters

For our first attempted optimization, we have chose to tune the spark-submit parameters and observe the various effects it has on the overall runtime performance of our model construction process.

No additional code-based optimization techniques have been employed in this process.

We systematically tested various configuration using an automated shell script for the following optimization parameters:

- Executor Nodes
- Executor Cores
- Memory (*Same Values Used for Master and Executors*)

Spark-Submit Performance Results

Baseline Runtime

We first ran our modeling code without any job submit or code optimizations using the following command:

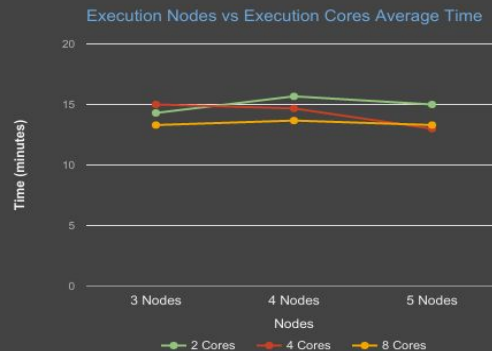
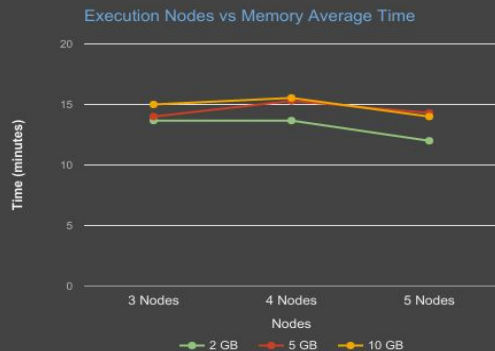
```
spark-submit target/scala-2.11/review_predict_2.11-0.1-SNAPSHOT.jar
```

Execution Time = **20 mins**

By default the system allocates two executors to perform the task.

Spark-Submit Performance Results

Nodes/Memory	3 Nodes				4 Nodes				5 Nodes			
Cores	2 GB	5 GB	10 GB	AVG	2 GB	5 GB	10 GB	AVG	2 GB	5 GB	10 GB	AVG
2 Cores	13 m	15 m	15 m	14.3 m	15 m	15 m	16 m	15.67 m	13 m	16 m	16 m	15 m
4 Cores	15 m	14 m	16 m	15 m	13 m	17 m	14 m	14.67 m	11 m	13 m	15 m	13 m
8 Cores	13 m	13 m	14 m	13.3 m	13 m	14 m	14 m	13.67 m	12 m	14 m	14 m	13.3 m
Average	13.67 m	14 m	15 m	14.2 m	13.67 m	15.3 m	14.67 m	14.54 m	12 m	14.3 m	15 m	13.76 m



Best Configuration: 5 Executor Nodes, 4 Executor Cores & 2 GB Memory
A 45% Decrease in Execution Runtime from Baseline Result

Persistence

To avoid redundant operations on RDD data, we have evaluated our overall RDD Lineage chain for any opportunities to introduce persistence into the picture. After finding the optimal location to insert persistence, we then experimented with various persistence types.

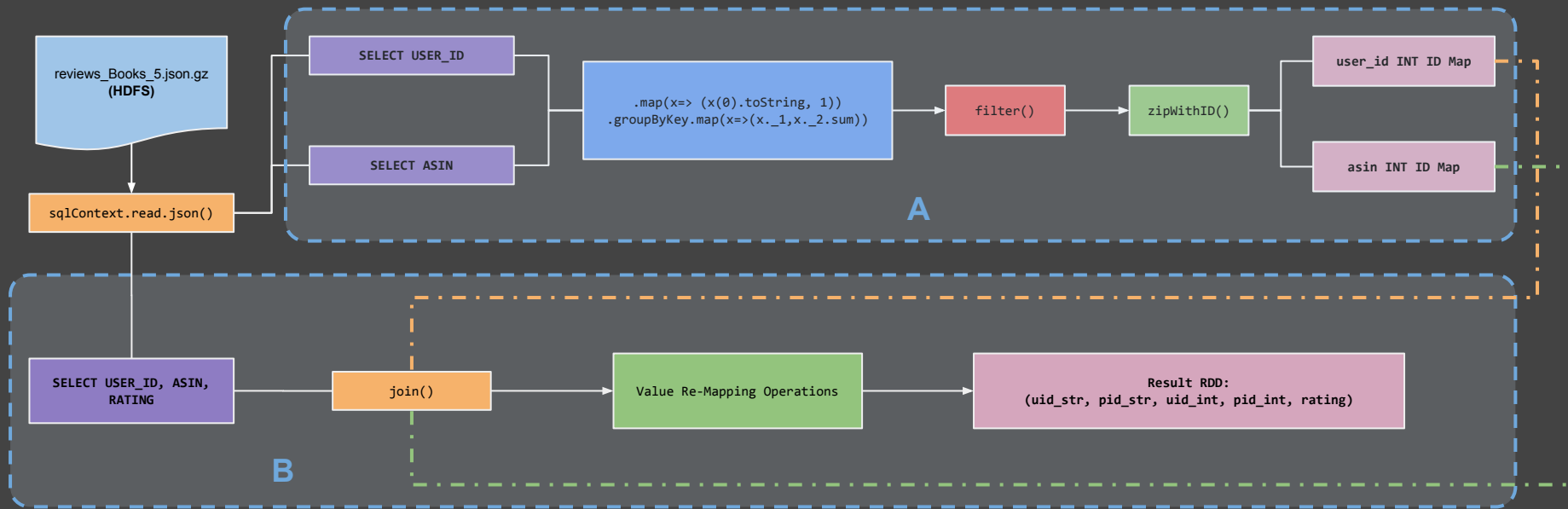
In our pipeline, we have persisted the corresponding training data RDD, as the internal source from MLlib makes multiple calls to this RDD. We utilized the various persistence types offered by Spark.

To perform our evaluation, we have kept *all job submission parameters consistent*.

Introducing Persistence

Looking back at our RDD Lineage chain, we can see that we have a branch between the two process.

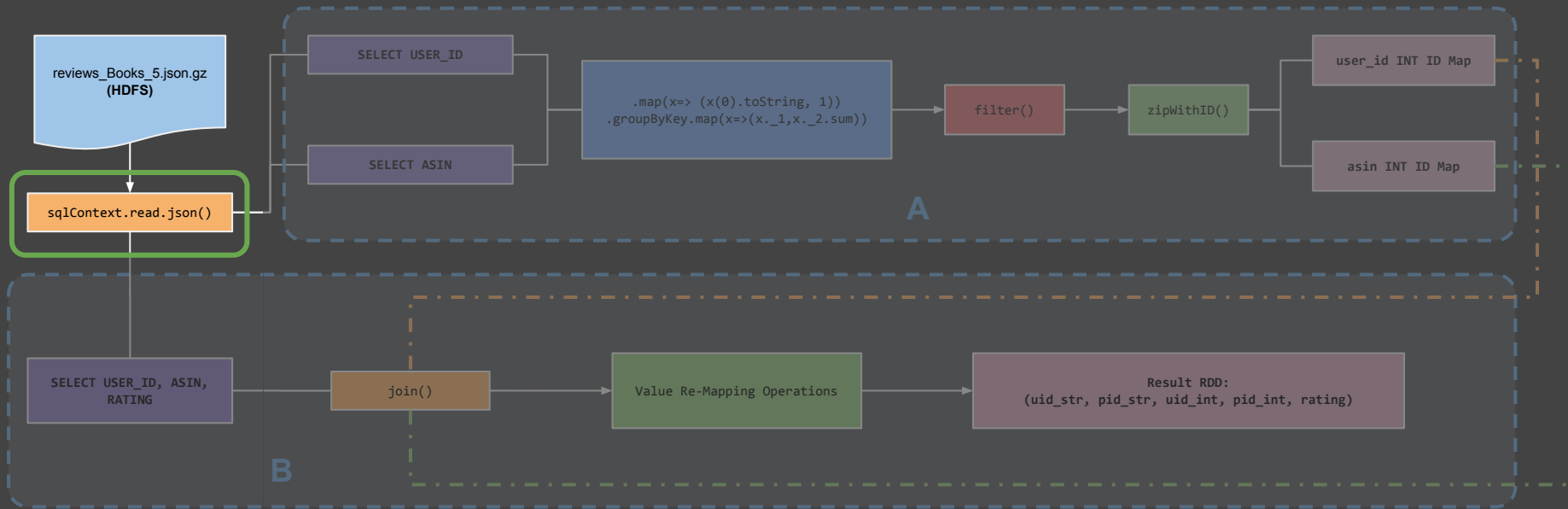
Introducing persistence prior to loading the file can be beneficial since *Disk I/O is very expensive to performance*.



Introducing Persistence

Looking back at our RDD Lineage chain, we can see that we have a branch between the two process.

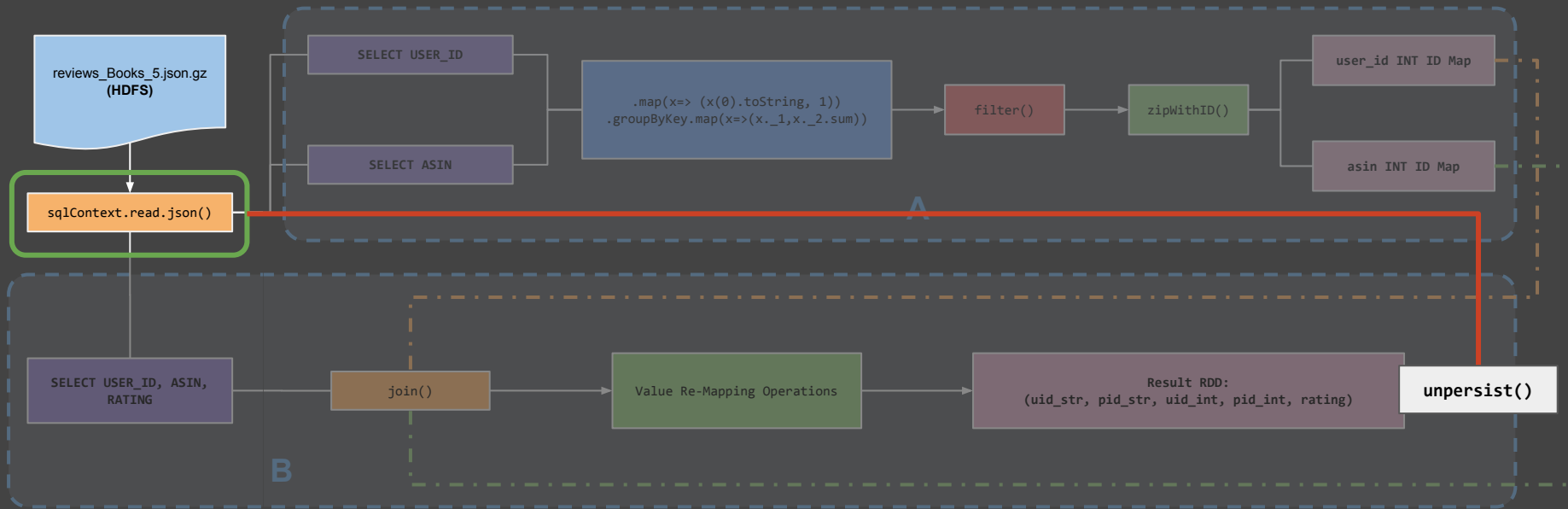
Introducing persistence prior to loading the file can be beneficial since Disk I/O is very expensive to performance.



Introducing Persistence

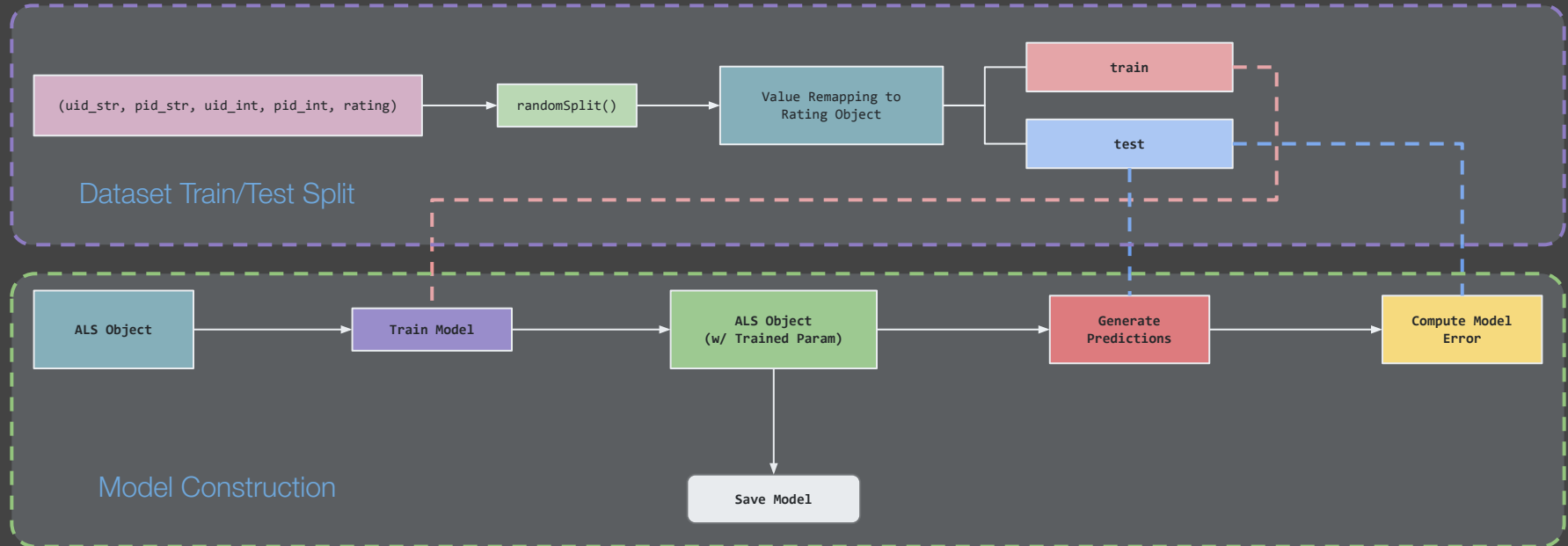
However, after transforming and reducing our dataset to a smaller RDD, it would probably be best to discard the persistence as there will probably be no need for it later down the road - *and to make room for other future persists*.

Therefore, we can **unpersist** it prior to completing this entire phase.



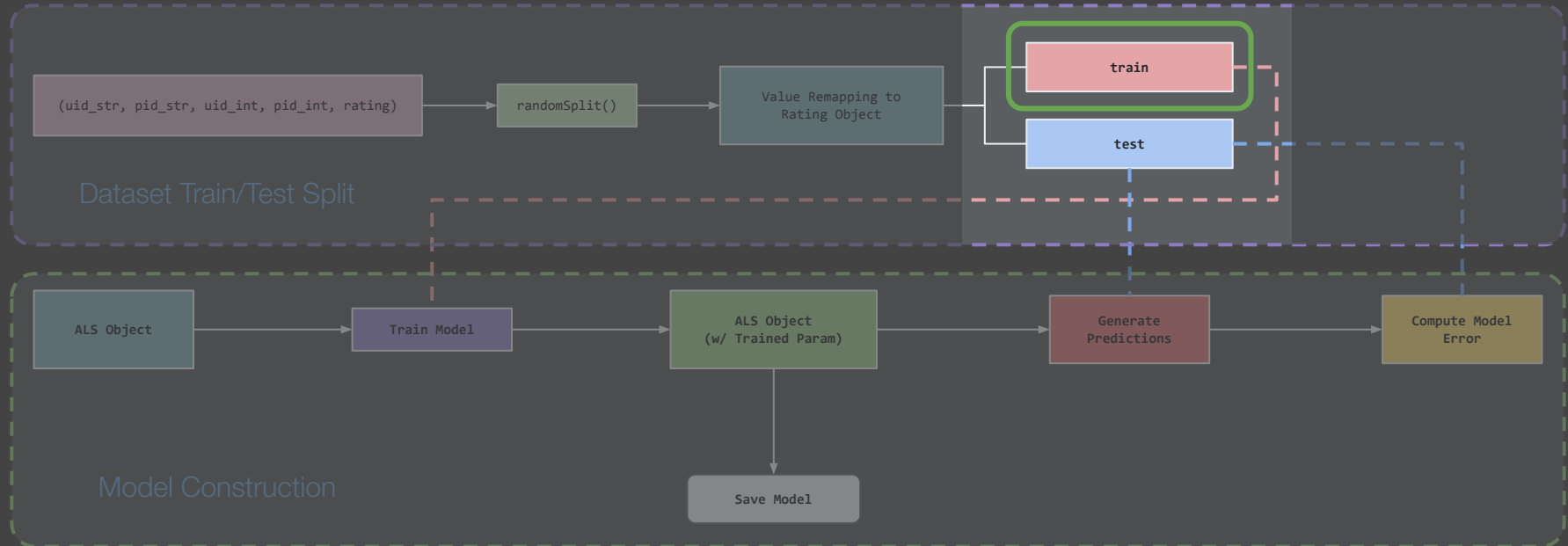
Introducing Persists

Another potential location to introduce persists on the training set used for our model - since the model will *often make calls to this data upon every iteration*.



Introducing Persists

Another potential location to introduce persists on the training set used for our model - since the model will often make calls to this data upon every iteration.



Introducing Persists

However, we have found this to be unnecessary, as MLlib's implementation for ALS already persists the data when you pass the data object in - thus a redundant action.

```
270     val userFactors = floatUserFactors
271       .mapValues(_.map(_.toDouble))
272       .setName("users")
273       .persist(finalRDDStorageLevel)
274     val prodFactors = floatProdFactors
275       .mapValues(_.map(_.toDouble))
276       .setName("products")
277       .persist(finalRDDStorageLevel)
```

Data Preprocessing Persistence Results

If we perform a comparative evaluation of the json parsing process, we can observe that persistence does help shed a small amount of the execution time. Notably we see a slight decrease in time for both Duration and GC Time

Below are the comparison metrics:

Before Adding Persistence

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	2.9 min	2.9 min	2.9 min	2.9 min	2.9 min
GC Time	0.5 s	0.5 s	0.5 s	0.5 s	0.5 s
Peak Execution Memory	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B
Input Size / Records	3.0 GB / 8898041	3.0 GB / 8898041	3.0 GB / 8898041	3.0 GB / 8898041	3.0 GB / 8898041

After Adding Persistence

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	2.8 min	2.8 min	2.8 min	2.8 min	2.8 min
GC Time	0.3 s	0.3 s	0.3 s	0.3 s	0.3 s
Peak Execution Memory	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B
Input Size / Records	3.0 GB / 8898041	3.0 GB / 8898041	3.0 GB / 8898041	3.0 GB / 8898041	3.0 GB / 8898041

Overall Persistence Results

Spark Submit Job Parameters

Parameters	Value
Driver Memory	10 GB
Executor Memory	10 GB
Number of Executors	5
Executor Cores	8

Experimental Runtime Results

Persistence Type	Runtime
NO PERSISTENCE	14 mins
DEFAULT	13 mins
MEMORY_ONLY	14 mins
MEMORY_AND_DISK	15 mins
MEMORY_AND_SER	14 mins
MEMORY_AND_DISK_SER	13 mins
DISK_ONLY	14 mins

Overall Result: Does not make much significant improvements to performance.

MLlib CF Parameter: Block Parallelization

Block Parallelization is a parameter offered by MLlib's ALS function which "defines the number of blocks users and items will be partitioned into in order to **parallelize computation**".

The default value, which can be set by using -1 uses the following method to determine how much blocks to use:

```
244     val numUserBlocks = if (this.numUserBlocks == -1) {  
245         math.max(sc.defaultParallelism, ratings.partitions.length / 2)  
246     } else {  
247         this.numUserBlocks  
248     }  
249     val numProductBlocks = if (this.numProductBlocks == -1) {  
250         math.max(sc.defaultParallelism, ratings.partitions.length / 2)  
251     } else {  
252         this.numProductBlocks  
253     }
```

However, we have attempted to experiment with a couple of different values to determine the overall effect in the performance runtime of our model development process.

Block Parallelization Results

Spark Submit Job Parameters

Parameters	Value
Driver Memory	10 GB
Executor Memory	10 GB
Number of Executors	5
Executor Cores	8

Experimental Runtime Results

Persistence Type	Runtime
-1 (DEFAULT)	14 mins
10	14 mins
25	18 mins
50	14 mins
100	13 mins

Overall Result: Slight improvement in overall performance.

Conclusions

In short, from our experiments we can conclude the following key findings and accomplishments:

1. Hardware & executor core scaling can help mitigate performance issues.
2. More Memory \neq Better Performance
3. Data Partitioning is helpful depending on the operation types performed.
4. We were able to cut down on runtime, on average, by **nearly 50%**.

Lesson Learned & Key Takeaways

- Always scope and narrow your task and objectives to specific items.
- Data/metric driven development is key in performance optimizations.
- Conduct multiple trials to obtain consistent and accurate measurements.
- Think outside the box and consider external factors not related to your code.
- Run/automate your experiments whenever possible, *especially during 3 AM - 7 AM*.

Performance optimization and evaluation is hard (*but pays off well if done correctly*).

