Quantcast

Apache Spark SQL optimizations for machine learning across internet-sized data

Michael Tong, Wenzhe Xu

Spark Data + AI Summit, 2021

May 26, 2021

Agenda

01 **Data transformations**
Simple transformations to reduce storage and compute costs.

02 **Leveraging natural features of datasets**
By understanding the distribution of your data you can make more intelligent decisions about how to store/use it.

03 **Low-level optimizations (pandas UDFs, numpy, JIT)**
A practical demonstration of some of the optimizations we apply on sample data

04 **(If time) Deep dive into the optimization engine**
A more open-ended discussion on some of the features of Spark query optimization and how to best utilize it.

Quantcast

01 | Data transformations

Quantcast

Data transformations

What data transformations did we look into?

- Lossless transformations. In other words, transformations such that we can perfectly reconstruct the original data from our transformed data.
- Why do this? Our data naturally comes in a messy format (strings of primarily csv format) and are naturally not very space efficient.
- These next few slides will discuss some of the basic transformations that we applied to our datasets.

Type casting

- The most commonly used transformation.
- If the fields are actually dates (datetime, timestamp, etc.), numbers (int, long, double, etc.), or some other type, actually specify that the type in the field (instead of string type)

Monotonically increasing ids

- Main idea is to use the monotonically_increasing_id function to create an index of unique value to long id and to store the long id value instead.
- Works best when there are <1M unique column values (allows for broadcast hash joins).
- Fields that we tend to use this trick on include user agent strings, browser types (like chrome, safari, etc.) and domains.

```
mono_ids = data\
    .select('col').distinct()\
    .withColumn('col_id', F.monotonically_increasing_id())
data\
    .join(F.broadcast(mono_ids), on='col')\
    .drop('col')
```

02 Leveraging natural features of datasets

Quantcast

Leveraging natural features of datasets

- First consider the set of queries that you tend to run over your data set (e.g., do you tend to queries on your dataset by id or some other column)
- Then consider if you often tend to often rely on collect or explode operations in order to run your queries.
- Often times, by considering these two points you can find optimization opportunities.

Using collect_set and collect_list to store data

- Main idea is that the easiest way to compress columnar data is to have less data.
- If you know that certain fields are:
  - commonly indexed against
  - natural ways to partition data (e.g. the column is an id)
- Then you can write queries like (data.groupby(partition_key).agg(collect_list(*other_fields))
- This trick typically can give us 10-25% smaller data footprints compared to round robin partitioning of our data.

Application: unique value counts

- If you input data is something like:
  - id, value
- You can apply something like the following to get:
  - id, [(value, count) tuples]

```
data\
    .groupby('id', 'value').count()\
    .withColumnRenamed('count', 'value_count')\
    .withColumn('value_count_pair', F.struct('value', 'value_count'))\
    .groupby('id')\
    .agg(F.collect_list('value_count_pair').alias('value_count_pairs'))
```

Application of collection: time series data

- If you input data is something like:
  - (id, time, transaction)
- You can apply something like the following to get:
  - (id, [(time, transaction) tuples])

```
data\
    .withColumn('time_transaction_pair', F.struct('time', 'transaction'))\
    .groupby('id')\
    .agg(F.collect_list('time_transaction_pair').alias('time_transaction_pairs'))
```

03 Low-level optimizations

Quantcast

Sample problem: introduction

- At Quantcast some of our models consider pairwise interactions between events.
- We have partitioned our data by *partition_key* such that there are only a few (20) events per partition.
- We want to use a machine learning model to give us information about all pairs of events per *partition_key* group.

Sample problem: data

```python
# parameters for sanitized data
n = 1000000
partition_size = 20
d = 20

# function that generates random numbers in the range of [-1, 1)
def generate_random_vector(size):
    return (2 * np.random.random(size)) - 1

# generate sample data
sample_data_1M = pd.DataFrame([[i // partition_size, i] for i in range(n)], columns=['partition_key', 'event_id'])
sample_data_1M['feature_vector'] = generate_random_vector((n, d)).tolist()

sample_data_1M.head()
```

	partition_key	event_id	feature_vector
0	0	0	[-0.5165411328143239, -0.7697248833258343, 0.2...
1	0	1	[-0.3785918054897246, -0.8154872951239256, 0.6...
2	0	2	[-0.6884332215238746, -0.532683039200122, 0.30...
3	0	3	[0.8309149840909955, -0.896161813570687, 0.511...
4	0	4	[0.7258664613993673, -0.1955426784967571, -0.8...

Sample problem: processing the data

```
# schema of our UDF
process_partition_schema = T.StructType([
    T.StructField('partition_key', T.IntegerType()),
    T.StructField('event_id1', T.IntegerType()),
    T.StructField('event_id2', T.IntegerType()),
    T.StructField('model_score', T.DoubleType()),
])

# processes a single partition_key of data
def process_partition(data):
    output = []

    # uses pandas iterrows to do the double for loop and score everything
    partition_key = data.iloc[0]['partition_key']
    for i, row1 in data.iterrows():
        event_id1 = row1['event_id']
        for j, row2 in data[i+1:].iterrows():
            event_id2 = row2['event_id']
            model_score = score_feature_pair(row1['feature_vector'], row2['feature_vector'])
            output.append([partition_key, event_id1, event_id2, model_score])

    return pd.DataFrame(output, columns=process_partition_schema.names)

sample_data_1M_df.groupby('partition_key').applyInPandas(process_partition, process_partition_schema)\
    .coalesce(1)\
    .write.mode('overwrite').parquet('/qfs/tmp/mtong/spark_demo_base')
```

Optimization idea: stop using pandas

- Ironically, our fastest pandas UDFs use barely any pandas.
- Many python libraries (built in ones, itertools, numpy) can be used to much greater effect than pandas
- For our UDF we can apply the following ideas:
  - Convert pandas to numpy types via *.values*
  - Use numpy vectors instead of lists for ML scoring
  - Use *enumerate* and *zip* instead of pandas iteration for speed
- We achieve a 6x speedup on our sample data (and in our real query too)

Optimization idea: stop using pandas

```
def process_partition(data):
    output = []

    # easy way to convert pandas series to numpy vectors and matrices
    event_ids = data['event_id'].values
    # convert to numpy matrix so each individual element is a numpy vector
    feature_matrix = np.array(data['feature_vector'].values.tolist())

    # use of enumerate and zip instead of using pandas iterrows
    partition_key = data.iloc[0]['partition_key']
    for i, (event_id1, feature_vector1) in enumerate(zip(event_ids, feature_matrix)):
        for event_id2, feature_vector2 in zip(event_ids[i+1:], feature_matrix[i+1:]):
            model_score = score_feature_pair(feature_vector1, feature_vector2)
            output.append([partition_key, event_id1, event_id2, model_score])

    return pd.DataFrame(output, columns=process_partition_schema.names)
```

Optimization idea: Scalar Pandas UDFs

- Main idea: what if we could process multiple partitions per UDF call at once?
- Need to use *collect_list* to aggregate rows per *partition_key*.
- Will require some data massaging and a repartition operation (due to optimization engine behavior)
- By using scalar pandas UDFs, we can process 10k partitions per python function call, allowing for batch optimization.
- This gives us a 10x speedup over the previous method

Optimization idea: Scalar Pandas UDFs

New UDF schema to process batches instead

```
process_partition_schema = T.StructType([
    T.StructField('partition_key', T.ArrayType(T.IntegerType())),
    T.StructField('event_id1', T.ArrayType(T.IntegerType())),
    T.StructField('event_id2', T.ArrayType(T.IntegerType())),
    T.StructField('model_score', T.ArrayType(T.DoubleType())),
])
```

Optimization idea: Scalar Pandas UDFs

Refactoring the UDF to process batches of data

```
# our first iteration of attempting pandas UDFs was to write a function for processing each row
def process_partition(partition_key, event_ids, feature_matrix):
    feature_matrix = np.array(feature_matrix.tolist())
    output = []
    for i, (event_id1, feature_vector1) in enumerate(zip(event_ids, feature_matrix)):
        for event_id2, feature_vector2 in zip(event_ids[i+1:], feature_matrix[i+1:]):
            model_score = score_feature_pair(feature_vector1, feature_vector2)
            output.append([partition_key, event_id1, event_id2, model_score])
    # massage columns so they are arrays of fields
    return list(zip(*output))

# and a separate function that could process all rows
def process_partition_batch(partition_key_series, event_ids_series, feature_matrix_series):
    results = [process_partition(*args) for args in zip(partition_key_series, event_ids_series, feature_matrix_series)]
    return pd.DataFrame(results, columns=process_partition_schema.names)

process_partition_udf = F.pandas_udf(process_partition_batch, process_partition_schema)
```

Optimization idea: Scalar Pandas UDFs

Refactored pyspark query to process data in batches

```
sample_data_1M_df\
    .withColumn('event_id_feature_pair', F.struct('event_id', 'feature_vector'))\
    .groupby('partition_key')\
    .agg(F.collect_list('event_id_feature_pair').alias('event_id_feature_pairs'))\
    .withColumn('partition_scores', process_partition_udf(
        'partition_key', 'event_id_feature_pairs.event_id', 'event_id_feature_pairs.feature_vector'))\
    .repartition(1)\
    .withColumn('partition_scores_zipped', F.arrays_zip(*[f'partition_scores.{col}' for col in process_partition_schema.names]))\
    .withColumn('partition_scores_exploded', F.explode('partition_scores_zipped'))\
    .select(*[F.col(f'partition_scores_exploded.{i}').alias(col) for i, col in enumerate(process_partition_schema.names)])\
    .write.mode('overwrite').parquet('/qfs/tmp/mtong/spark_demo_scalar_udf')
```

Optimization idea: batching

- Main idea: Now that we’re batching things, can we write our expensive operations as batched ones?
- This works well for scalar pandas UDFs. The general idea is:
    - For each pandas UDF call
    - Group all of your expensive data into a single vector
    - Process your data as a vector
    - Split the data up back into individual rows to return.

Optimization idea: JIT

- Main idea: Can we run compiled functions?
- Short answer: yes you can.
  - I personally like numba/JIT
  - Only really gives you performance improvements over numpy types (vectors, matrices, etc.)
  - Generally takes much more development time to produce JIT-optimized functions.
  - Sometimes barely helps, sometimes up to 5-10x speedup. Depends heavily on particular use case.

04 Deep-dive into the optimization engine

Quantcast

First: Learn how to read the SQL tab

- Allows you to understand how Spark is trying to run your query.
- Among other useful things it gives you:
  - A visual representation of Spark's query plan
  - A text representation of Spark's query plan
  - Spark's current progress on your query
  - Useful information on how many rows and how large are various intermediate steps.

Pyspark filter and project operations

- In general, Spark will try to push filter operations as far up the query plan as it possibly can.
- This is useful except when computing the filter operation is expensive (i.e., the filter function depends on an extremely expensive UDF.

Pyspark project operations on structs

- As a general rule of thumb, try to avoid directly calling specific fields of structs.
- Especially if those structs were generated via expensive functions.
- Under certain circumstances Spark may decide that the best way to process your query is to generate the structs multiple times if you select multiple individual columns from a single struct.
- This can usually be circumvented by either:
  - Writing the intermediate result to a temporary location.
  - Adding a sort/shuffle operation between the UDF and project operations

Pyspark struct field behavior

- In general, it is difficult to efficiently use structs with UDFs if:
  - You end up processing multiple columns from the struct differently downstream.
  - You end up using filter operations on only a subset of the struct fields.
- To deal with these problems, my recommendations are to:
  - Rethink how you're processing your data. Often times structs are not necessary.
  - If you really insist on using structs, pay careful attention to what spark is doing by reading the query plans closely.

05 Conclusions

Quantcast

Conclusions

- When trying to scale Spark to large datasets, there are several avenues of optimization including:
    - Data transformations of key fields.
    - Changing schemas and partitioning rules of data sets.
    - Optimizing UDFs to apply custom logic on these transformed data sets efficiently.
    - Deeply understanding the optimization engine to bring it all together.

Links

Repository with the slides and the source code:

https://github.com/mrtong96/spark_2021_talk

Questions?

Quantcast

Thank you

Quantcast