Questions from the Chat

Q: Why do we see many stages/tasks as skipped? Is it because we had cached a dataframe in the code?
A: Indeed, it could mean that data has been fetched from cache and it doesn't need to re-execute the given stage. It can also happen when reusing shuffle files.
A not on shuffle files: when performing a wide transformation, e.g. *group by* or *join*, data has to be shuffled across the executors. To do so, Spark will have to write data into temporary shuffle files. When having to redo the exact same group by or join, Spark can bypass it by simply reading those shuffle files directly, instead of doing the full shuffle again.

Q: Could you describe if at the level above (Spark Cluster UI) is possible to see more than 1 spark application running? Somehow when submitting applications through Microsoft Data Factories, the end result is 1 application DatabricksShell running, as opposed to multiple submissions to the same cluster?
A: As far as I know this is not possible, for now, to have multiple Spark applications with different Spark contexts running at the same time on a single cluster in Databrick. They will all use the same Spark context.

Q: Can you please suggest an alternative to explode?
A: Here are some examples of transforming complex data types including other alternatives to explode.
But remember: if you need to create multiple lines out of one containing specific values in an array, then the explode function is exactly what you need. I'm simply saying that we should avoid using explode as long as we can since it will drastically increase the size of the data on RAM.
We can mitigate it though by reducing the size of our Spark partitions when reading data by lowering the value for configuration parameter *spark.sql.files.maxPartitionBytes*. The idea is to proactively create smaller Spark partitions so that when the explode function is applied, our partitions will grow in size that will still be manageable.
If you have access to the upstream source's code, you could also update it so that you have one line per value, and not one line with an array of values. But then the size of the data will increase on the upstream side, and this might not make sense from a business perspective. This is a tradeoff that you will have to think about.

Q: How to use the Spark UI on a cluster when multiple jobs are running?
A: The Spark UI can indeed become tricky/hard to use with lots of Spark jobs running, potentially at the same time. The simplest way is probably to set the job description for each job to make the search way easier, just like it's done with all experiments in the Spark UI Simulator.
Example: *sc.setJobDescription("Step A: Basic Initialization")*

Q: Link to all Delta's optimizations?
A: https://docs.microsoft.com/en-us/azure/databricks/delta/optimizations/

Q: Can you please share a document which helps determine partition sizing wrt to cluster size?
A: Most of the time you will want to have partitions be less than 200MB, irrespective of your cluster size. And it's mostly about determining the correct number of shuffle partitions compared to the actual size of the partitions. I highly recommend watching Daniel Tomes' Spark Summit video on Apache Spark Core - Deep Dive - Proper Optimization. I put the link at the section related to your question, but watching the full video is highly recommended.

Q: Broadcast variable/table will unpersist automatically after job execution or it has to be run explicitly?
A: It will eventually be garbage collected, but if you can, unpersist to free sooner?

Q: Difference between broadcast and broadcast hash join?
A: A broadcast hash join is one (physical) way to run a join, i.e. one algorithm to run a join. We can use a "broadcast hint" to tell Spark that this type of join should be prioritized (compared to a sort-merge join for instance). We can also use the built-in broadcast function to mark a DataFrame as small enough for use in "broadcast joins". The concept of "broadcast"in Spark is a bit larger though, and not dedicated only to DataFrames. See the Spark documentation on Broadcast Variables for more information. Long story short, you can broadcast read-only values (e.g. maps, integers, lists) on each executor so that you incur the shipping costs only once instead of everytime a task needs it. The same word is used in both cases because we essentially do the same thing: send a copy of the entire value/DataFrame to each executor only once and then reuse it.

Q: What is the downside of AQE given it's not enabled by default?
A: AQE is enabled by default since Spark 3.2. The need to manually disable AQE is very rare to be honest. It might be useful when you're 100% sure that you don't want to coalesce partitions and keep only a given number of shuffle partitions across your job, or when you don't want to broadcast DataFrames. But those cases are really context/data specific.

Q: Does AQE work on top of Catalyst Optimizer?
A: Yes. Catalyst applies before runtime to find the best physical plan possible. AQE applies during runtime once we have started running the physical plan and that we have gathered runtime statistics (e.g. size of data written to intermediate files).

Q: Does this mean AQE improves performance over time considering it feeds in the RDD to logical plan. And we might not see improved performance right from the start?
A: Yes. Catalyst optimizer already does the execution planning, so if AQE is ON it will dynamically adjust the plan at runtime.

Q: Does AQE work with cached datasets?
A: It depends on the scenario that gets created at runtime.
Example 1: for a join, if a DataFrame is already cached, then Spark knows its size and can therefore apply a broadcast hash join (if it's small enough) when creating the physical plan

instead of waiting for AQE to kick in (there is no need to wait for AQE in that case since we already know the size).
Example 2: we are in the same configuration as in Example 1, except the cached DataFrame is too big to be broadcasted by default, but we want to filter it before applying the join, and when filtered it becomes small enough to be broadcasted. In that case AQE will kick in.

Q: As I understand, AQE will be largely guided by the metadata collected at the stage creation. Is this understanding correct?
A: Yes it does use the Runtime statistics but also adjust the actual execution, which can happen even after stage creation. Read more here.

Q: I saw in the code that *spark.sql.adaptive.skewjoin.enabled* is enabled. Is it necessary for skewed join optimization with AQE?
A: Yes, this parameter is used to enable/disable skewed join optimizations.

Q: So, AQE cannot increase the number of partitions. That's why we should keep the number of partitions higher initially?
A: Yes and no. Theoretically, AQE can actually increase the number of partitions. To do so, make sure AQE is enabled and set *spark.sql.shuffle.partitions* to *"auto"*. BUT this value is not the default for now as we're still benchmarking it internally. In the case you don't want to get into edge cases, you can indeed set a very high number for *spark.sql.shuffle.partitions* and then let AQE deal with finding the correct number of shuffle partitions with that value as an upper bound.

Q: Since AQE is enabled by default, should we still look into join optimization, skews, and shuffle partitions explicitly?
A: With AQE turned on, these optimizations are already taken care of, so you should not necessarily spend too much time on this. However, sometimes there could be issues such as these. I would still want to spend time on the skew issue to try to understand where the problem comes from. Is it something unexpected? If yes then we have data issues that need to be solved right away. Is it something expected that we can handle easily? E.g. there are lots of null or zero values that we can filter out before performing the join/aggregation? Then we should handle this, as it's easy and will definitely improve the runtime performance. Is it something expected that we cannot handle easily? Then let AQE deal with it.

Q: Is there an objective reason to choose Python over R apart from Databricks focusing support on Python?
A: Broadly speaking there is wider community support for Python than R and in effect more wider adoption. However, R is still a great choice for purely statistical data analysis. There are also a few performance considerations when compared to python in the context of Spark. Also, with Python you get the support to build great UI based apps. For R, the options are a bit limited and more costly.

Q: When making the jdbc call, you can set a where clause as port of the dbtable parameter. Does that method behave the same way (predicate pushdown) as using the .filter() call?

A: After testing this, the physical plans will be different but the output read from the database will be the same, meaning predicate pushdown has been applied. In case we use the dbtable parameter, there will not be any PushedFilters, but the table source will be different: it will be the sql query containing the where clause, not the table name. In both cases the filtering part will be done by the database, not Spark.
NB: based on the API there might be separator parameters to pass on predicates. You can also turn on and off this functionality (predicate pushdown) in the JDBC config. The default value is true, in which case Spark will push down filters to the JDBC data source as much as possible.

Q: What does Delta cache do?
A: See the Optimize performance with caching documentation for more information. Quite simply, it will create copies of remote files in nodes' local storage using a fast intermediate data format. Success reads of the same data are then performed locally, which results in significant improved reading speed. See this table for a summary of the differences between Delta caching and Spark caching.

Q: What resources (compute, memory) will Auto Optimize use for example with 100GB of data?
A: This is rather hard to quantify, but Auto Optimize will run with the same cluster as the actual transaction happening on the delta.

Q: Is there some limit on the number of columns in a delta table for which it will collect statistics?
A: Not exactly. By default, Delta will collect statistics on the first 32 columns defined in your table schema. But you can change this value using the table property delta.dataSkippingNumIndexedCols. Adding more columns to collect statistics would add more overhead as you write files.

Q: Why do we need Z-ordering when we have Partitioning available in Delta?
A: Partitioning is to "partition" our dataset into smaller subsets so that we only read what we need. Partitioning should be used on columns of low cardinality (else we create too many folders and small files which has a dramatic impact when scanning and reading files), e.g. like country or year/month. With ZOrder, Delta Lake automatically lays out the data in the files based on the column values, and uses the layout information (i.e. the columns statistics retrieved when writing data to the files) to skip irrelevant data while querying. This has great benefits if you know that a specific group of columns will be heavily used. ZOrder works best with columns of high cardinality and needle-in-the-haystack queries.

Q: Can we use an instance pool in job clusters?
A: yes: see, the instancePoolId property.

Q: Predicate pushdown is an internal spark operation. If we want it differently, how can we control that?
A: I'm sorry but I don't understand the question