Risks of Using Partitioning

A major risk when using partitioning is creating partitions that lead you into the small files problem. When this happens, partitioning a table will actually worsen query performance (the opposite of the goal when using partitioning) because it causes too many small files to be created. This is more likely when using dynamic partitioning, but it could still happen with static partitioning—for example if you added a new partition to a sales table on a daily basis containing the sales from the previous day, and each day's data is not particularly big.

When choosing your partitions, you want to strike a happy balance between too many partitions (causing the small files problem) and too few partitions (providing performance little benefit). The partition column or columns should have a reasonable number of values for the partitions—but what you should consider *reasonable* is difficult to quantify.

Using dynamic partitioning is particularly dangerous because if you're not careful, it's easy to partition on a column with too many distinct values. Imagine a use case where you are often looking for data that falls within a time frame that you would specify in your query. You might think that it's a good idea to partition on a column that pertains to time. But a TIMESTAMP column could have the time *to the nanosecond,* so every row could have a unique value; that would be a terrible choice for a partition column! Even to the minute or hour could create far too many partitions, depending on the nature of your data; partitioning by larger time units like day, month, or even year might be a better choice.

As another example, consider the default.employees table on the VM. This has five columns: empl_id, first_name, last_name, salary, and office_id. Before reading on, think for a moment, which of these might be reasonable for partitioning (assuming the table will eventually be much larger than the five rows in our sample table)?

- The column empl_id is a unique identifier. If that were your partition column, you would have a separate partition for each employee, and each would have exactly one row. In addition, it's not likely you'll be doing a lot of queries looking for a particular value, or even a particular range of values. This is a poor choice.

- The column first_name will not have one per employee, but there will likely be many columns that have only one row. This is also true for last_name. Also, like empl_id, it's not likely you'll need filter queries based on these columns. These are also poor choices.
- The column salary also will have many divisions (and even more so if your salaries go to the cent rather than to the dollar as our sample table does). While it may be that you'll sometimes want to query on salary ranges, it's not likely you'll want to use individual salaries. So salary is a poor choice. A more limited salary_grades specification, like the ones in the salary_grades table, might be reasonable *if* your use case involves looking at the data by salary grade frequently.
- The office_id column identifies the office where an employee works. This will have a much smaller number of unique values, even if you have a large company with offices in many cities. It's imaginable that your use case might be to frequently filter your employee data based on office location, too. So this would be a good choice.

You also can use multiple columns and create nested partitions. For example, a dataset of customers might include country and state_or_province columns. You can partition by country and then partition those further by state_or_province, so customers from Ontario, Canada would be in the country=ca/state_or_province=on/ partition directory. This can be extremely helpful for large amounts of data that you want to access either by country or by state or province. However, using multiple columns increases the danger of creating too many partitions, so you must take extra care when doing so.

The risk of creating too many partitions is why Hive includes the property hive.exec.dynamic.partition.mode, set to strict by default, which must be reset to nonstrict before you can create a partition. (See the note about this, near the end of the “Loading Data Using Dynamic Partitioning” reading.) Rather than automatically and mechanically resetting that property when you're about to load data dynamically, take it as an opportunity to think about the partitioning columns and maybe check the number of unique values you would get when you load the data.