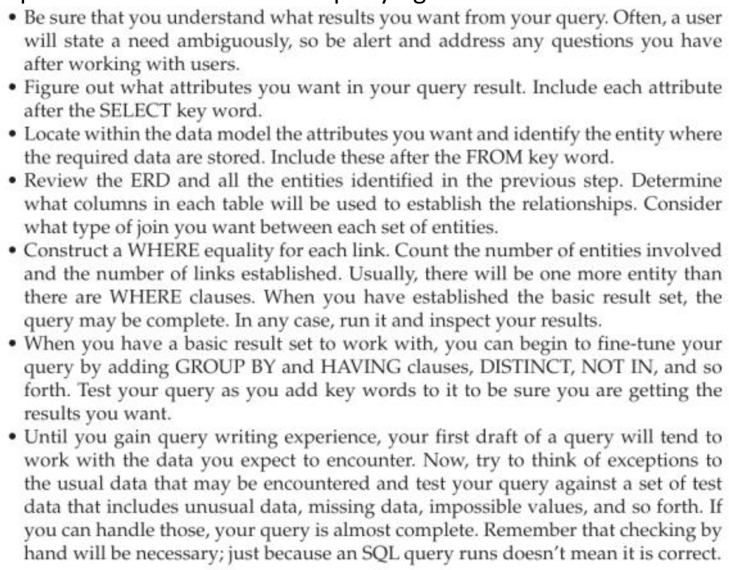
TIPS FOR DEVELOPING QUERIES

SQL's simple basic structure results in a query language that is easy for a novice to use to write simple ad hoc queries. At the same time, it has enough flexibility and syntax options to handle complicated queries used in a production system. Both characteristics, however, lead to potential difficulties in query development. As with any other computer programming, you are likely not to write a query correctly the first time. Be sure you have access to an explanation of the error codes generated by the RDBMS. Work initially with a test set of data, usually small, for which you can compute the desired answer by hand as a way to check your coding. This is especially true if you are writing INSERT, UPDATE, or DELETE commands, and it is why organizations have test, development, and production versions of a database so that inevitable development errors do not harm production data.

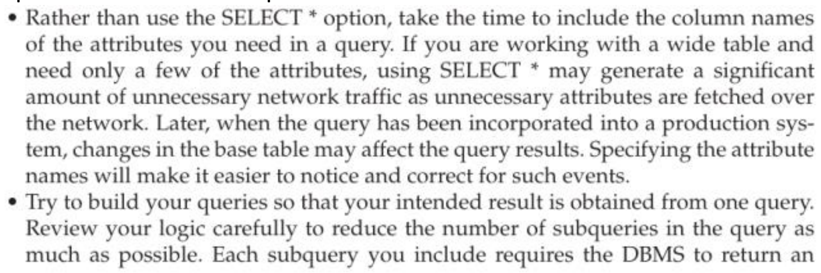
Things that could go wrong:
As a novice query writer, you might find it easy to write a query that runs without error. Congratulations, but the results may not be exactly what you intended. Sometimes it will be obvious to you that there is a problem, especially if you forget to define the links between tables with a WHERE clause and get a Cartesian join of all possible combinations of records. Other times, your query will appear to be correct, but close inspection using a test set of data may reveal that your query returns 24 rows when it should return 25. Sometimes it will return duplicates you don't want or just a few of the records you want, and sometimes it won't run because you are trying to group data that can't be grouped. Watch carefully for these types of errors before you turn in your final product.

Tips for better results when querying:



- Be sure that you understand what results you want from your query. Often, a user will state a need ambiguously, so be alert and address any questions you have after working with users.
- Figure out what attributes you want in your query result. Include each attribute after the SELECT key word.
- Locate within the data model the attributes you want and identify the entity where the required data are stored. Include these after the FROM key word.
- Review the ERD and all the entities identified in the previous step. Determine what columns in each table will be used to establish the relationships. Consider what type of join you want between each set of entities.
- Construct a WHERE equality for each link. Count the number of entities involved and the number of links established. Usually, there will be one more entity than there are WHERE clauses. When you have established the basic result set, the query may be complete. In any case, run it and inspect your results.
- When you have a basic result set to work with, you can begin to fine-tune your query by adding GROUP BY and HAVING clauses, DISTINCT, NOT IN, and so forth. Test your query as you add key words to it to be sure you are getting the results you want.
- Until you gain query writing experience, your first draft of a query will tend to work with the data you expect to encounter. Now, try to think of exceptions to the usual data that may be encountered and test your query against a set of test data that includes unusual data, missing data, impossible values, and so forth. If you can handle those, your query is almost complete. Remember that checking by hand will be necessary; just because an SQL query runs doesn't mean it is correct.

Tips for more efficient queries:



- Rather than use the SELECT * option, take the time to include the column names of the attributes you need in a query. If you are working with a wide table and need only a few of the attributes, using SELECT * may generate a significant amount of unnecessary network traffic as unnecessary attributes are fetched over the network. Later, when the query has been incorporated into a production system, changes in the base table may affect the query results. Specifying the attribute names will make it easier to notice and correct for such events.
- Try to build your queries so that your intended result is obtained from one query. Review your logic carefully to reduce the number of subqueries in the query as much as possible. Each subquery you include requires the DBMS to return an

interim result set and integrate it with the remaining subqueries, thus increasing processing time.

- Sometimes data that reside in one table will be needed for several separate reports. Rather than obtain those data in several separate queries, create a single query that retrieves all the data that will be needed; you reduce the overhead by having the table accessed once rather than repeatedly. It may help you recognize such a situation by thinking about the data that are typically used by a department and creating a view for the department's use.

Finals tips:

1. ***Understand how indexes are used in query processing*** Many DBMSs will use only one index per table in a query—often the one that is the most discriminating (i.e., has the most key values). Some will never use an index with only a few values compared to the number of table rows. Others may balk at using an index for which the column has many null values across the table rows. Monitor accesses to indexes and then drop indexes that are infrequently used. This will improve the performance of database update operations. In general, queries that have equality criteria for selecting table rows (e.g., WHERE Finish = "Birch" OR "Walnut") will result in faster processing than queries involving more complex qualifications do (e.g., WHERE Finish NOT = "Walnut") because equality criteria can be evaluated via indexes. You will have an opportunity to revisit indexing in Chapter 8.
2. ***Keep optimizer statistics up to date*** Some DBMSs do not automatically update the statistics needed by the query optimizer. If performance is degrading, force the running of an update-statistics-like command.
3. ***Use compatible data types for fields and literals in queries*** Using compatible data types will likely mean that the DBMS can avoid having to convert data during query processing.
4. ***Write simple queries*** Usually the simplest form of a query will be the easiest for a DBMS to process. For example, because relational DBMSs are based on set theory, write queries that manipulate sets of rows and literals.
5. ***Break complex queries into multiple simple parts*** Because a DBMS may use only one index per query, it is often good to break a complex query into multiple, simpler parts (which each use an index) and then combine together the results of the smaller queries. For example, because a relational DBMS works with sets, it is very easy for the DBMS to UNION two sets of rows that are the result of two simple, independent queries.
6. ***Don't nest one query inside another query*** Usually, nested queries, especially correlated subqueries, are less efficient than a query that avoids subqueries to produce the same result. This is another case where using UNION, INTERSECT, or MINUS and multiple queries may produce results more efficiently.
7. ***Don't combine a table with itself*** Avoid, if possible, using self-joins. It is usually better (i.e., more efficient for processing the query) to make a temporary copy of a table and then to relate the original table with the temporary one. Temporary tables, because they quickly get obsolete, should be deleted soon after they have served their purpose.
8. ***Create temporary tables for groups of queries*** When possible, reuse data that are used in a sequence of queries. For example, if a series of queries all refer to the same subset of data from the database, it may be more efficient to first store

this subset in one or more temporary tables and then refer to those temporary tables in the series of queries. This will avoid repeatedly combining the same data together or repeatedly scanning the database to find the same database segment for each query. The trade-off is that the temporary tables will not change if the original tables are updated when the queries are running. Using temporary tables is a viable substitute for derived tables, and they are created only once for a series of references.

9. ***Combine update operations*** When possible, combine multiple update commands into one. This will reduce query processing overhead and allow the DBMS to seek ways to process the updates in parallel.
10. ***Retrieve only the data you need*** This will reduce the data accessed and transferred. This may seem obvious, but there are some shortcuts for query writing that violate this guideline. For example, in SQL, the query SELECT * from EMP will retrieve all the fields from all the rows of the EMP table. But if the user needs to see only some of the columns of the table, transferring the extra columns increases the query processing time.
11. ***Don't have the DBMS sort without an index*** If data are to be displayed in sorted order and an index does not exist on the sort key field, then sort the data outside the DBMS after the unsorted results are retrieved. Usually, a sort utility will be faster than a sort without the aid of an index by the DBMS.
12. ***Learn!*** Track query processing times, review query plans with the EXPLAIN command, and improve your understanding of the way the DBMS determines how to process queries. Attend specialized training from your DBMS vendor on writing efficient queries, which will better inform you about the query optimizer.
13. ***Consider the total query processing time for ad hoc queries*** The total time includes the time it takes the programmer (or end user) to write the query as well as the time to process the query. Many times, for ad hoc queries, it is better to have the DBMS do extra work to allow the user to more quickly write a query. And isn't that what technology is supposed to accomplish—to allow people to be more productive? So, don't spend too much time, especially for ad hoc queries, trying to write the most efficient query. Write a query that is logically correct (i.e., produces the desired results) and let the DBMS do the work. (Of course, do an EXPLAIN first to be sure you haven't written "the query from hell" so that all other users will see a serious delay in query processing time.) This suggests a corollary: When possible, run your query when there is a light load on the database because the total query processing time includes delays induced by other load on the DBMS and database.