Running Head: JOINS AND SUBQUERIES

SQL Statements: Joins and Subqueries

Jered McClure  
Walden University

SQL Statements: Joins and Subqueries

The primary advantage of a join and/or nested query is the ability to consolidate information without permanently denormalizing a database. In essence, using a join or nested query is the equivalent of a temporary localized denormalization of a database, which does not affect the operational capacity of the entire entity set which makes up that database. Also, it allows for the gathering of information from various tables based on specific rules and/or joins which are not natural in terms of how the database is formed. For instance, given two separate tables which have no foreign keys to each other, if a data link exists between a column in the first table and a column in the second table, a join can be made regardless of any built in keys.

However, the downside to this setup is in the data sizes brought back from such joins. Normalization is done in order to limit the amount of data each table contains, thereby increasing overall functionality of the tables. Creating select statements with joins and/or subqueries can be detrimental to this performance boost. That being said, this generally only occurs over large data sets, or where the join is done incorrectly causing an out of control Cartesian product (e.g. a many-to-many join where the number of rows in the first table is multiplied by the number of rows in the second table). Additionally, this can occur when the joins and subqueries take place over many levels, thereby causing multiple statements to run within one session.

Joins are generally used over nested queries when the attributes of one table and the attributes of another table coincide or link based off of given keys or candidate keys. For example, if a column in TABLE\_A coincides to a column in TABLE\_B regardless of whether that column is a built in key in either table, then the situation would call for a JOIN statement.

An Example

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Table\_1** | | |  | **Table\_A** | | |
| **Column1** | **Column2** | **Column3** | **ColumnA** | **ColumnB** | **ColumnC** |
| 1 | a | a1 | a1 | blah | a |
| 2 | a | a3 | b1 | blech | e |
| 3 | b | a1 | c1 | bloop | i |
| 4 | b | c2 | a2 | blarb | o |
| 5 | c | b2 | b2 | blop | u |
| 6 | c | c2 | c2 | blurp | y |
| 7 | d | b1 | a3 | bleep | null |

Neither of these two tables is keyed against the other in the database schema. However, Column3 in Table\_1 and ColumnA in Table\_A are clearly a match for one another that should bring back a data set of some sort if a JOIN were made using those two columns.

**SELECT** T1.COLUMN1,

T1.COLUMN2,

TA.COLUMNA,

TA.COLUMNB,

TA.COLUMNC

**FROM** TABLE\_1 **AS** T1

**INNER** **JOIN**

TABLE\_A **AS** TA

**ON** T1.COLUMN3 = TA.COLUMNA

Which would return:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| COLUMN1 | COLUMN2 | COLUMNA | COLUMNB | COLUMNC |
| 1 | a | a1 | blah | a |
| 2 | a | a3 | bleep | null |
| 3 | b | a1 | blah | a |
| 4 | b | c2 | blurp | y |
| 5 | c | b2 | blop | u |
| 6 | c | c2 | blurp | y |
| 7 | d | b1 | blech | e |

A subquery, on the other hand, is used when very specific data must be aggregated and returned for a decision to be made within the query. An example of this is, given a large set of customers, we want to know how many customers there are of each customer type, and what their overall income is based on new field which we want to add to the query called PivotA.

**SELECT** CUST\_TYPE,

**COUNT**(CUSTOMER\_NO) **AS** CUST\_COUNT,

**SUM**(AGGREGATE\_COUNT) **AS** PivotA

**FROM**(

**SELECT** CUSTOMER\_NO,

CUST\_TYPE

**CASE** **WHEN** CUST\_TYPE **BETWEEN** 1 **AND** 5 **THEN** 1000

**WHEN** CUST\_TYPE **BETWEEN** 6 **AND** 10 **THEN** 2000

**ELSE** 3000 **END** **AS** AGGREGATE\_COUNT

**FROM** CUSTOMER) **AS** AGGREGATE\_TABLE

**GROUP** **BY** CUST\_TYPE

**ORDER** **BY** CUST\_TYPE

This simple query allows for data to be added to a set of data without actually altering any tables in the database. Also, because this data is not needed for the operation of the database, it can be kept in this dynamic form thereby alleviating any storage requirements in the database itself.

In other words, if the goal of the query is to link tables together in order to return a denormalized version of the data set then use a join. However, if the goal is to modify, or bring back an aggregate transformation of the data sets, then a sub query may be called for. This being said, always remember that doing either of these two statements will cause a slowdown in the overall database operation. This is, of course, based on the overall size of the data set, and the speed of the database.

Reference

Coronel, C., Morris, S., & Rob, P. (2012). *Database Systems: Design, Implementation, and Management* (10th ed.). Boston, MA, USA: Cengage Learning.