Adding Subqueries to MySQL, or What Does it Take to Have a Decision-Support Engine? *

Antonio Badia, Matt Chanda, Bin Cao

Computer Engineering and Computer Science Department
University of Louisville
Louisville, KY 40292

ABSTRACT
MySQL is an OLTP system without the ability to handle SQL subqueries. As part of our project in query optimization, we add subquery processing to MySQL through rewriting. First, we incorporate the ability to handle subqueries in the FROM clause; then rewrite queries with subqueries in the WHERE clause by moving the subquery to the FROM clause and introducing suitable conditions to link the subquery and the main query tables. The result is a complete and correct unnesting procedure for SQL queries. The idiosyncrasies of SQL semantics make some rewritings tricky. While optimization issues are not directly addressed, the rewriting opens up the possibility of more efficient query processing in several cases.

Categories and Subject Descriptors
H.2.4 [Database Management]: Systems—Query Processing

General Terms
Languages, Algorithms.

Keywords
SQL, query optimization, unnesting.

1. INTRODUCTION
Since Kimball’s proclamation that “one size does not fit all” ([9]), it is customary to divide database systems into transaction processing (OLTP) and Decision-Support (OLAP) systems. The differences in several key parameters of the workload, like type of queries, size of tables... have convinced experts that different systems must be used for optimal performance in each environment: OLAP systems require specific techniques for data warehousing, like materialized views, query rewriting, etc. which implement research results specifically geared towards Decision-Support environments ([2, 8]). However, vendors of existing database systems have not developed new systems from scratch. Eager to exploit their existing technology (and the investment of time and money), they have fine-tuned their products, introducing specific data warehousing tools into their offering. While this has required substantial extensions, it has also shown the enormous flexibility in the underlying relational technology.

It is fair to say that one of the most salient characteristics (even though not the only one) of a Decision-Support system is the complexity of queries and the size of databases supported. This makes query processing and optimization an area which requires special attention when designing an OLAP system. Thus, one of main obstacles for a non-OLAP system to become one is to develop extremely efficient support for complex queries. The question is, what does it take to achieve such support? The authors faced this question recently when they tried to add support for subqueries to the MySQL system ([12]). While well known and widely used, MySQL is considered a light system, with a small footprint, light resource usage, and the ability to handle heavy transaction loads (and thus very popular as a back-end for Web sites), but unable to handle Decision-Support style SQL queries. In fact, MySQL does not handle nested subqueries in its current incarnation1. In this project, we set up to add subquery processing to MySQL while avoiding large changes to the query processing code. Our strategy followed two steps: first, we added the ability to handle subqueries in the FROM clause. Second, we rewrote queries with subqueries in the WHERE clause by moving the subquery to the FROM clause and changing the conditions in the WHERE clause. While the changes are trivial in some cases (e.g., it is well known that an IN condition is equivalent to a semijoin),

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DOLAP ‘02, November 4–9, 2002, McLean, Virginia, USA.
Copyright 2002 ACM 1-58113-492-4/02/0011 ...$5.00.

*This research was sponsored by NSF under grant IIS-0091928. The opinions expressed herein are solely those of the authors. A full version of this paper is available as a technical report at date.spd.louisville.edu/badia/forloop.html.

1Adding subquery support to MySQL has been in the developers to-do list for quite some time, but it has not materialized at the time of writing this paper. Part of the reason for the postponement is probably the fact that MySQL does not use a real query tree and its back-end processing is geared towards SPJ (Select-Project-Join) queries. Thus, the addition of subquery processing support would require deep changes in the query processing mechanism.
some other cases required more complex changes. The tricky (and sometimes confusing) semantics of SQL make some of the translations difficult to define and prove correct. In the process, we show that all SQL queries, no matter how complex, can be supported by a query engine with support for projection, selection, join, semijoin, outerjoin, and grouping (union and difference of queries can be handled recursively). Of course, this is not all the system needs; support for large files, special indices (bitmaps, index joins, etc.), materialized views, and more, is needed for a true OLAP system. For instance, an efficient sorting method is also required for the \textsc{sort by} clause of SQL, and as implementation tool for other algorithms. In this paper, we restrict our attention to the set of algebraic operators that is required to support SQL queries. We stress that our aim is to obtain correct, complete transformations for SQL subqueries (i.e. transformations applicable to all kinds of subqueries). Issues of optimization are mentioned only from an algebraic perspective, but the larger issue of whether the transformation provide performance advantages is left for further work, since a large part of performance depends on physical characteristics (support for large files, special indices, etc.). Thus, we focus on changes needed to the back-end from an algebraic perspective. The conclusion is that the difference between support of all SQL queries and SPJ queries is not as significant as one would think; the main difference lies in the optimizer\footnote{A different issue is whether SQL itself is well suited to OLAP. It is by now a well established opinion that SQL needs to be extended and complemented for a true Decision-Support environment ([6, 9]).}.

In section 2 we describe how MySQL processes queries, and explain the difficulties of adding subquery support in the standard way. In section 3 we show our strategy to deal with subqueries in the \textsc{from} clause, and in section 4, with \textsc{where}-clause subqueries. In section 5, we analyze the trade-offs of our approach and discuss optimization issues, and in section 6 we offer some conclusions.

2. Overview of MySQL Query Processing

MySQL uses a client-server model with multiple threads on the server. When the database receives a SQL query, it spawns a new thread to handle the query and sends the text of the SQL query to the parser. The parser is a large lex-yacc script which constructs a query structure corresponding to the query. This structure basically contains all strings found in the SQL query classified by clause. The parser simply checks for correct SQL grammar; it does very little verification of the query’s elements (the parser will not check the existence of the tables, the existence of columns or if an aggregate function was used without a GROUP BY clause). The query structure from the parser is passed to the query processor. The first step in the query processing is to check the user’s access to all the tables in the table list of the query. The tables need to be opened and locked so the query can execute. Once the tables are opened and locked, the processor moves to a select (query) handler. If a query is a \textsc{union}, a recursive \textit{union handler} will be called. Otherwise, the query handler is called to execute a simple query. This handler will carry out the query by interaction with the back-end processor, which takes care of file (disk) access.

After analyzing MySQL’s processing flow and data structures, it seems clear that MySQL does not use a query tree, i.e. a structure where the query is represented with a relational algebra-like expression which can be manipulated for optimization and further processing. The query is represented by two lists: a list of table (file) accesses, called the \textit{table list}, and a list of joins, called the \textit{join list}. Selections are attached to the tables to which they apply, and so are part of the table list. Optimization seems to be absent from the system except at the most trivial level. In particular, and as a result of the representation chosen for the query, it seems that all selections are pushed down and pipelined regardless of selectivity or form; and that join order is pretty much fixed by the query. No different orderings are considered; choices for join algorithm seem limited to nested loop (with or without use of indices in the inner relation). It seems clear that this framework is sufficient to handle the type of queries present in transaction-oriented environments (simple SPJ queries, usually involving one or few tables, and often with highly restrictive selections), while offering low overhead and simplicity. However, the framework is ill-suited for Decision-Support queries; it is in particular difficult to determine how to best extend it to deal with subqueries. The absence of a real query tree makes implementation of standard processing techniques, like unnesting ([11, 10, 3]), extremely complicated\footnote{On a practical note, the almost non existent documentation on system internals and abundant use of global variables makes any modifications to the source code quite risky.}. Thus, the approach taken was determined by the goal of changing as little of the code as possible, and the need to work without a query tree.

3. Adding From Clause Subqueries

The ability to deal with \textsc{from} clause subqueries was added by modifying only the front-end. The changes were designed to make the back-end believe it was processing a regular query when it was handling the subquery. Details on the modifications, as well as examples, are given in the technical report of which this paper is a summary ([1]). A simple example is provided here to illustrate the process: assume a database with tables \texttt{table1}((\texttt{cola, colb}) and \texttt{table2}((\texttt{cola, colc}). Consider the query in 1(a). The query has a FROM-clause subquery in it. When the parser detects it, the subquery is sent for processing through the parser and all the way to the back end, which is instructed to deposit the results in a temporary table. This table is associated with its alias (a). All the information about the table is then added to the table list for the main query before proceeding with the processing of the main query. Thus, the back end sees the queries in 1(b) and in 1(c). By the time this last query is processed, an entry for table a has been created and is pointing to the file where the results (temporary table) have been stored.

Even though the idea is quite simple, there were many areas that needed to be changed to add \textsc{from} clause subqueries to MySQL. The parser had to be modified to both accept the additional functionality in the SQL language and to store it correctly in the query structure. The query structure was expanded to store the information concerning the subquery. The query processor had to be modified to actually execute the subquery and incorporate it into the query structure again so that the back-end could finish the processing of the
SELECT cola FROM (SELECT cola, colb FROM table1 WHERE colb < 4) AS a WHERE a.cola > 3

(a) Original Query

SELECT cola, colb FROM table1 WHERE colb < 4 WHERE a.cola > 3

(b) Subquery

SELECT cola FROM a WHERE a.cola > 3

(c) Final Query

Figure 1: Transformation for FROM clause subqueries

The whole process was designed to be recursive, and therefore it can handle queries with a FROM clause subquery which in turn contains a FROM clause subquery, and so on. Once subquery processing is done, processing of the main query continues. The back-end does not know that the temporary table created by the subquery is not an actual table in the database; it knows that the table has an alias and that it does not have any indices. The subquery processing is contained in its own thread. After the query finishes processing, the temporary table has to be released and the thread terminated.

4. ADDING WHERE CLAUSE SUBQUERIES

We classify SQL subqueries corresponding to the condition that they express. Thus, we distinguish between EXISTS and NOT EXISTS subqueries, IN and NOT IN subqueries, SOME subqueries, ALL subqueries and aggregate subqueries. Each one of these classes is further subdivided according to whether the subquery is correlated or not. We present each class of queries separately; however, a general strategy takes care of all cases with just two variations.

Our strategy was to move the subquery in the WHERE clause to the FROM clause, and then use the temporary table created to finish up processing. In general, two different procedures were developed: for some cases, the table that results from the subquery must be connected to other existing tables through an outer join. For other cases, the connection must be established through an outer join. In all cases, the subtleties of SQL semantics (especially involving nulls and repeated tuples) call for a careful rewriting of the conditions relating subquery results and main query tables. Even cases considered simple (like IN and EXISTS predicates) needed careful modification.

We describe the transformations with pairs of patterns, i.e. combinations of constants (keywords) and variables that specify the form the rewrite takes (keywords are shown in all uppercase; variables are shown in lowercase). The first pattern in the pair shows the original query, and the second one the result of the rewriting. Use of the same variables in the rewritten pattern shows how elements of the old query are used. Square brackets ([,]) are used to show optional elements. Thus, [a | b] is used to choose one of a or b. In general, this is used for transformations which apply to correlated and non-correlated queries. Parenthesis are used for clarity. A different pattern pair is developed for each case. In the WHERE clause of a query, we show explicitly the predicate that connects query and subquery (called the linking predicate), together with any attributes or operators involved. We also show explicitly any predicate introducing correlation in the subquery (i.e. if such predicate is absent, it is understood that the subquery is not correlated). Variable names are meant to be descriptive; thus op and op2 stand for operators in predicates; single-col stands for an attribute name which appears alone in a SELECT clause; corr-value appears for an attribute name that introduces correlation in a subquery; and aggr-column denotes the aggregate function and attribute name used in aggregates subqueries. Variables ending in list or list2 are meant to stand for a list of elements.

In our transformations, we have made the following assumptions:

- each relation has an attribute denoted by # which serves as a (non null) primary key; and
- all queries are connected, i.e., all tables appearing in a FROM clause are joined together in a common table. That is, there are no Cartesian products in the original query.

The authors believe that both assumptions are quite common in practice and hence do not detract from the applicability of our results.

4.1 Joined Subqueries

The queries that can be transformed into join are EXISTS and NOT EXISTS non-correlated subqueries; IN and SOME subqueries (both correlated and non correlated); and non-correlated, aggregated subqueries. We show each case separately.

1. [NOT] EXISTS non-correlated Subqueries. The EXISTS function checks for the existence of a result from a subquery. In non correlated subqueries, this result is independent of the main query; thus, this case reduces to checking whether a given query returns the empty set or not. Thus, a very simple transformation can take care of the non-correlated EXISTS. The subquery can simply be inserted in the FROM clause. As a result, a Cartesian product takes place between the tables in the main query and the result of executing the subquery. If such a result is empty, so is the Cartesian product (since, for any relation \( R \), \( R \times \emptyset = \emptyset \)), and therefore no result is returned (as it would happen in the original query). On the other hand, if the table representing the result of processing the subquery is not empty, the Cartesian product is not empty either, but multiple copies of the tuples in the main query would result. Therefore, we use the following trick: we retrieve an arbitrary constant which does not appear in the database instead of any value. The end result is to retrieve either one tuple or nothing for the subquery. This allows existence testing and gets rid of...
the duplicate problem. In our implementation, we follow this approach since it can be made very efficient, with some simple additions. The additions include a Boolean variable to hold the result of the subquery, and a boolean flag to let the subquery processor know that the subquery was in an EXISTS function. If it is, the processor sets the row limit on the query to one row and executes the query into a temporary table. It then checks the number of rows in the temporary table. If one row exists then the value of the exists subquery result in the subquery class is set to true. Thus, no Cartesian product is actually performed in practice. We show the approach in figure 2(b). In our implementation, if the NOT keyword is added before the EXISTS function, the correct answer can be obtained by inverting the flag. However, the approach cannot be extended to NOT EXISTS in SQL, so we propose the transformation shown in figure 2(c) for the general case. In this approach, we use counting to determine the number of rows in the resulting table. Whenever the number is 0, the EXISTS predicate is false and the NOT EXISTS predicate is true. Whenever the result is not 0, the EXISTS predicate is true and the NOT EXISTS predicate is false. Thus, we use $\text{SQ.count}(*) \neq 0$ for EXISTS and $\text{SQ.count}(*) = 0$ for NOT EXISTS. Note our use of '*' instead of any attribute name, in order to make sure that the transformation is correct even in the presence of nulls in column-list2.

2. IN/SOME Subqueries. This case is well known from the literature in query optimization through unnesting ([10, 5, 3, 11]). However, it presents some interesting technical problems. For years, it was considered that both predicates were essentially equivalent to a join. But IN and join are not equivalent, since a join may introduce multiple copies of a given tuple, while the predicates IN is simply true or false for a given tuple, no matter how many matches this tuple has in the subquery result. In fact, the IN case is equivalent to a semijoin, which is not part of many implementations of SQL. In the semijoin, because of the implicit projection, no duplicates are allowed. The same is true of the SOME predicate. In the case of IN, we can solve this problem in SQL by adding the DISTINCT keyword to the SELECT clause of our subquery, so as to remove duplicates. In effect, when a join is later performed with tuples in the main query, those tuples have at most one match and therefore no duplication occurs. This translation is shown in figure 3(b). However, this does not work for SOME, since the condition in the SOME may cause a tuple to find multiple matches in the join even if duplicates are removed (essentially, IN and SOME are equivalent, so IN is just a special case of SOME). Thus, as a general solution we use a GROUP BY clause to remove duplicates after the join between the tables in the main query and the result of evaluating the subquery. Note the addition of a key from one of the tables in table-list indicated

5The row limit of a query is the maximum number of rows that will be returned by the database for a query, and is usually undefined. This row limit is also set for aggregate subqueries, see next subsection.

SELECT column-list
FROM table-list
WHERE criteria-list AND [EXISTS | NOT EXISTS] (SELECT column-list2
FROM table-list2
WHERE criteria-list2)

(a) Original Query

SELECT column-list
FROM table-list,
(SELECT e FROM table-list2
WHERE criteria-list2)
WHERE criteria-list

(b) Translated Query (EXISTS case)

SELECT column-list
FROM table-list,
(SELECT COUNT(*) FROM table-list2
WHERE criteria-list2) AS SQ(CT)
WHERE criteria-list AND SQ.CT ![= | =] 0

(c) Translated Query (general case)

Figure 2: Transformation for Non-correlated EXISTS/NOT EXISTS Subqueries

by one-of-the-table-list.#. The transformation is shown in figure 3(c).

Again, the parts in brackets are there for the correlated case; they should be ignored for the non correlated case.

3. Non-correlated Aggregate Subqueries. In this case, an aggregation in the SELECT of the subquery forces it to return a single value. For this case, we move the subquery to the FROM clause and join the resulting table to the tables in the main query. Note that, since the result of the subquery is guaranteed to have only one row, no duplicates can be introduced. The translation is shown in figure 4.

Note also that we have transformed the condition into a join even though we know that temporary table SQ has only one row with one column. Thus, in practice we can think of this as a selection; however, this is the only way to rewrite the query in SQL. In our implementation, we again set the row limit to 1, and this improves performance significantly by holding the returned result in main memory and avoiding a real join.

4.2 Outerjoin Subqueries

Queries that need an outerjoin and some further transformation of the conditions are the correlated EXISTS and NOT EXISTS subqueries; subqueries with NOT IN or ALL (both correlated and non-correlated); and correlated subqueries with aggregation. Again, we present each case separately.
SELECT column-list
FROM table-list
WHERE criteria-list AND
parent-tbl.col [IN | op1 SOME]
(SELECT single-col
FROM table-list2
WHERE criteria-list2)

(a) Original Query

SELECT column-list
FROM table-list
(SELECT DISTINCT single-col
FROM table-list2
WHERE criteria-list2) AS SQ
WHERE criteria-list AND
parent-tbl.col [= | op1] SQ.single-col
AS Q

(b) Translated Query (IN case)

SELECT column-list
FROM (SELECT column-list, one-of-table-list.#
FROM table-list,
(SELECT single-col,[table.column]
FROM table-list2
WHERE criteria-list2) AS SQ
WHERE criteria-list AND
parent-tbl.col [= | op1] SQ.single-col
[AND SQ.table.column op2 corr-value]
GROUP BY one-of-table-list.#, column-list)
AS Q

(c) Translated Query (general case)

Figure 3: Transformation for IN/SOME Subqueries

1. Correlated [NOT] EXISTS subqueries. Correlated NOT EXISTS subqueries cannot be dealt with in a manner similar to non-correlated ones. The reason is that in the non correlated case a subquery was either empty or not regardless of any condition involving values in the main query, while now a subquery may be empty because the correlated value finds no match in the subquery. It is not possible to express this negation (or, equivalently, universal quantification) directly in SQL; instead, we paraprhase the negation using the outer join operator. This is the same strategy we will follow in the negation of \( \text{IN} \) (the NOT \( \text{IN} \) operator) and in the universal quantifier (ALL).

While the idea is very simple, implementing it in SQL calls for solving some issues raised by nulls and by the duplication introduced by the outer join. The first step in the translation for the NOT EXISTS function is to add the value in the subquery that is compared to the correlated attribute to the SELECT clause of the subquery, The subquery is then moved to the FROM clause and the subquery result is outerjoined to the correlated value table. The rows that do not match up in the left outer join are desired, so we count the values in the outer table after grouping by one of the key values. Note that in this case we count on the attribute specified by the original query, so that nulls (whether introduced by the padding of the outer join or present in the original data) are ignored. Note also that the grouping has the effect of removing duplicates from the final result. The translation is shown in figure 5.

2. NOT IN/ALL Subqueries. The idea to deal with this case is simple. Unfortunately, the idiosyncrasies of SQL semantics make it a tricky case. Our general strategy for dealing with negation (or, equivalently, universal quantification) is to paraphrase the logical equivalence \( \neg \exists x \varphi(x) = \forall x \neg \varphi(x) \) with the help of the outer join operator as follows: an outerjoin based on a condition which is the negation of the original condition in the NOT \( \text{IN} \) or ALL predicates is used, and then the tuples which are padded with nulls are selected. Thus, to evaluate the predicate \( \text{att} > \text{ALL (Select \ att2 ...)} \), we outerjoin the table containing \( \text{att} \) with the table that results from evaluating the subquery using the comparison operator \( \leq \), and pick the tuples that do not match anything (i.e. if it is never the case that \( \text{att} \leq \text{att2} \), then \( \text{att} > \text{att2} \) for all values of \( \text{att2} \)). Unfortunately, this simple approach fails in the presence of nulls: if the attribute \( \text{att2} \) contains nulls, the ALL predicate will fail, but so will all comparisons with the negated operator, thus qualifying the original tuple. Also, the predicate returns false when \( \text{att} \) is null, except when the subquery returns an empty answer, in which case the predicate returns true. Therefore, in our translation we must take care of these situations, which we do by modifying the condition of the outer join as follows: if any of the attributes involved in the outer join condition (\( \text{att} \) and \( \text{att2} \), in our ex-
SELECT column-list
FROM table-list
WHERE criteria-list AND [EXISTS | NOT EXISTS]
(SELECT column-list2
FROM table-list2
WHERE criteria-list2 AND
  table.column op corr-value)

(a) Original Query

SELECT column-list
FROM (SELECT column-list, one-of-table-list1.#
  FROM table-list LEFT OUTER JOIN
    (SELECT column-list2, table.column
     FROM table-list2
     WHERE criteria-list2) AS SQ
  ON ((parent-tbl.col [NOT IN | op1 ALL]
    SQ.single-col
    OR parent-tbl.col IS NULL
    OR SQ.single-col IS NULL)
    [AND SQ.table.column op2 corr-value])
  WHERE criteria-list)
AS Q

(b) Translated Query

Figure 5: Transformation for Correlated EXISTS/NOT EXISTS Subqueries

ample) is null, we still qualify the tuple as matching. Note that the outer join of relation $R$ and an empty relation results in all tuples in $R$ padded with nulls; and therefore we would pick them all. This coincides with the semantics of the original query, since when the subquery evaluates to an empty result, all rows in the main query qualify. The behavior of $\text{NOT IN}$ is similar: $\text{att NOT IN } Q$, where $Q$ is a subquery, will be false if $\text{att}$ is null, unless $Q$ evaluates to an empty answer, in which case the predicate evaluates to true (even if $\text{att}$ is null).

The translation, shown in figure 6) will move the subquery to the FROM clause, but will move the correlation criteria to the outer join condition (whenever a correlation exists. This pattern is to be used for both correlated and non-correlated subqueries, using or not the part in brackets) and complement it as explained above. Thus, $\text{neg-op1}$ is meant to denote the negation of $\text{op1}$ (i.e. $\leq$ for $>$, etc.).

3. Aggregate Subqueries. The case of correlated subqueries with aggregation is probably the best known and most studied in the literature since it originated the zero count bug of Kim’s approach ([5]). Our translation follows the approach of using an outerjoin before computing the aggregate and grouping, as suggested in ([3, 11]). This computation is moved to the FROM clause, as in the other cases. The query is then finished by executing the linking condition in the WHERE clause of the main query. The transformation is shown in figure 7. Also, whenever the aggregate being computed is $\text{COUNT(*)}$, we replace it by $\text{COUNT(one-of-tables.#)}$ in the translated query, in order to avoid the zero count bug.

5. OPTIMIZATION ISSUES

Our approach is motivated by having a complete and correct mechanism to handle SQL subqueries. Thus, we cover all types of SQL subqueries, while most rewriting approaches (including unnesting) do not have complete coverage. Also, we deal with null values and repeated tuples, to make sure that the semantics of the original queries are respected. However, the purpose of rewriting is usually optimization, and therefore the question must be asked as to whether the present approach presents advantages from a performance point of view. In general, it is obvious that the proposed approach presents a serious issues with respect to optimization: by moving subqueries from the WHERE to the FROM clause, we are still dividing the work in two parts dictated by the SQL query. Traditional unnesting merges operators beyond the boundaries of query and subquery, and therefore provides a greater degree of optimization. However, in the present case there are good reasons for considering the approach taken as a sensible alternative. From the point of view of implementation in MySQL, the absence of a query tree makes such algebraic transformations extremely hard to implement. The approach followed allowed us to localize all changes in the front-end. Moreover, we point out that a query with a FROM clause subquery can be considered as a single query with no sub-
**SELECT** column-list  
**FROM** table-list  
**WHERE** criteria-list AND  
parent-tbl.col op1  
(SELECT aggr-column  
FROM table-list2  
WHERE criteria-list2 AND  
table.column op2 corr-value)  

(a) Original Query  

**SELECT** column-list  
**FROM** table-list,  
(SELECT aggr-column,  
corr-value-table.# AS #  
FROM corr-value-table  
LEFT OUTER JOIN table-list2  
ON (table.column op2 corr-value)  
WHERE criteria-list2  
GROUP BY corr-value-table.#) AS SQ  
WHERE criteria-list AND  
parent-tbl.col op1 SQ.aggr-column AND  
one-of-table-list.# = SQ.#  

(b) Translated Query  

---

Figure 7: Transformation for Correlated Aggregate Subqueries

queries from an algebraic point of view (i.e. it is similar to the addition of an assignment operator to the relational algebra, which does not increase expressive power and is mainly a matter of convenience). Thus, if a query tree is added to MySQL’s query processing, our rewriting would make it very easy to actually carry out such transformations. Finally, complex unnesting is known to limit the amount of rewriting possible, because of the user of outer joins (but see [4] or [7] for some solutions). Therefore, the additional advantage of unnesting over the present approach is limited.

Nevertheless, it is clear that further possibilities exist for optimization. For instance, in all cases trivial subqueries (no WHERE or GROUP BY clause) should not generate a subquery in the FROM clause; the table name should suffice (projection -without duplicate removal- can be harmlessly moved around). There are some additional opportunities for improvement on particular cases.

For **join cases**, when an **EXISTS** subquery is done, all we need to do is check whether the answer (temporary table) is empty. If so, no further processing is necessary, as the answer to the query is going to be empty too. If the temporary table is not empty, the temporary table can be disposed of and the main query can be executed with disregard to the **EXISTS** subquery. Thus, one could stop processing as soon as one tuple is generated for the subquery result. However, this is not expressible in a purely algebraic framework. For non-correlated aggregates, an additional optimization is to take the value that is returned from the subquery and create a constant to remove the additional join from the query plan -this is also not expressible in an algebraic framework, but it is carried out by many commercial query processors. Also, if the constant returned is a null, there is no need for further processing, as all comparisons will fail and the final result is going to be empty. Therefore, if the value is computed first, there is a possibility for further optimization. Note that this is equivalent to pushing down the **EXISTS** predicate (or the aggregated subquery) to be done always first, despite the fact that it may have a high cost associated with it. Therefore, this plan should be considered as one more alternative for the query optimizer, which should estimate a cost for it and compare it to other plans.

For **outerjoin cases**, an interesting possibility arises. In the cases of **NOT IN**, **ALL** and non-correlated **EXISTS** and **NOT EXISTS**, we note that after the outer join we are going to keep only rows where the value is null (which we locate by counting the number of non-null matches), this operation really is equivalent to a LEFT ANTIJOIN\(^6\). Thus, a system that is equipped with algorithms for ANTIJOIN could execute this query directly. The advantage of this approach is that even though the algorithm itself may have a cost in the same order as an algorithm for join, we would expect the output size to be smaller.

For the **ALL** case, there is also an opportunity for additional optimization. It should be noted that if nothing passes **criteria-list2** then every tuple will return true for the **ALL** predicate; therefore in the non-correlated case no further processing of the predicate is needed, while in the correlated case if we get an empty answer for a certain value of the correlation there is no need for further testing on that particular group.

Finally, we point out for some cases our transformation turns out to be basically equivalent to well known unnest algorithms. However, for some cases not covered by previous rewrites, the approach proposed is bound to outperform traditional tuple-at-a-time (also known as nested loop) processing.

---

6\(^*\)Because of nulls, one must be careful with the definition of ANTIJOIN. Here, the ANTIJOIN of relations \(R\) and \(S\) on condition \(A \theta B\) is defined as the set of tuples \(t\) in schema \(sch(R) \cup sch(S)\) such that (1) \(t[R]\) corresponds to a tuple \(t’\) in \(S\) such that there are no tuples in \(R\) satisfying the join condition for \(t’\), and \(t[R]\) is padded with nulls; or (2) \(t[R]\) corresponds to a tuple \(t’\) in \(R\) such that there are no tuples in \(S\) satisfying the join condition for \(t’\), and \(t[S]\) is padded with nulls. Left (right) antijoin is restricted to tuples in condition 1 (2).
optimizer must support. Of course, further abilities (like support for CUBE and other extensions ([6])) are also required for a true OLAP engine. The main obstacle to our approach turns out to be the idiosyncrasies of the SQL language, which make some transformations tricky.

We are considering other types of rewriting, which are left for further work. In its current incarnation, our implementation in MySQL includes all non-correlated subqueries. We are currently exploring the work needed to extend the approach to correlated subqueries, and to allow a complete unnesting of all queries. Unfortunately, the MySQL code is not conducive to extensive changes; thus such project is still under development.

7. REFERENCES


