Efficient Computation of Subqueries in Complex OLAP

Michael O. Akinde
Swedish Meteorological & Hydrological Institute
Folkborgsvägen 1,
SE-60364 Norrköping, Sweden
michael.akinde@smhi.se

Michael H. Böhlen
Aalborg University
Fredrik Bajers Vej 7E,
DK-9220 Aalborg, Denmark
boehlen@cs.auc.dk

Abstract

Expressing complex OLAP queries involving nested expressions using normal group-by, aggregation, and joins can be extremely difficult. This paper proposes a technique that translates nested query expressions into an algebra extended with a complex OLAP operator. The GMDJ is an operator with a simple and easy to optimize implementation that is particularly useful for OLAP computations because the size of intermediate results is bound by the size of the base-value argument relation. We show that all SQL subqueries can be expressed in the algebra using GMDJs. This not only makes it easy to integrate subqueries into any query engine that supports GMDJs, but also gives access to a broad range of OLAP optimization strategies for evaluating subqueries. We discuss the coalescing of GMDJs and the completion of tuples, two GMDJ optimizations that are particularly relevant to subquery processing. Our experimental results demonstrate the validity and efficiency of our approach for computing subquery expressions.

1 Introduction

On-line analytical processing (OLAP) has been and remains the focus of intense research and commercial activity. Large private and public organizations use data warehouses to store and organize information collected in their standard business processes. The need to analyze such data has led to a significant body of research concerned with multidimensional querying and analysis of very large databases. New application areas and technology have led to diversification in the types of data collected, and analysis of such data may often be highly complex and ad-hoc.

Features like `CUBE BY` [16] and grouping sets [11] allow the user a more fine-grained control over grouping and aggregation. Such features allow the user to express complex OLAP operations that would be difficult, or impossible, to express using SQL. Another feature often encountered in complex OLAP are subqueries. It is often easier to express a complex OLAP query using subquery constructs than to construct equivalent flattened queries. An analyst considering the usage of an IP network might wish to analyze network statistics by asking questions such as: "On an hourly basis, what fraction of the traffic originating from IPs for which there exist a user account is due to web traffic?" While conveniently formulated with the help of subqueries, such queries are generally a hard problem for query optimization.

The naive approach to handling subqueries is to use nested-loops (tuple iteration semantics). However, this method is generally inefficient [17] and does not leave much scope for optimization. As a result, the research community has sought to discover methods that remove nested subqueries by unnesting or flattening these queries into SQL or relational expressions without the nesting. Section 1.1 briefly surveys these methods.

1.1 Related Work

The majority of research on subqueries has, directly or indirectly, been based on the work of Kim [17] or Ceri and Gottlob [8]. Kim's algorithm, and the extensions developed by Ganski and Wong [15], and Muralikrishna [19, 20] are source-to-source transformations over SQL and were based on a join, or an aggregate then join/outer-join approach.

Ceri and Gottlob [8] defined the semantics of SQL in terms of relational algebra and converted nested SQL queries into a relational algebra extended with aggregate functions and using set difference. Bækgaard and Mark [5] built on this work, developing a nested algebra for describing subqueries, but without considering aggregates, duplicates, or null values.

Dayal [12] studied the transformation of `EXISTS` subqueries into a variant of the relational algebra that handles duplicate rows. His transformations employ a variety of join, outer-join and join-aggregate constructs for the unnesting. Bültzingsloewen [6] presented a two-step join unnesting algorithm that transformed SQL expressions with subqueries into a calculus and an algebra with aggregation.

More recent work [24, 23, 14] has built on these techniques to addresss problems such as aggregation, duplicates, and NULL values in the data.

Rao and Ross [23] developed a strategy for reusing invariants in the evaluation of nested queries. Their strategy is useful for queries where conventional unnesting fails to provide efficient performance and thus supplements, rather than replaces, other algorithms. The techniques described by Rao and Ross can be leveraged for the GMDJ query processing used in this paper, and thus allow us to generalize these techniques for general subquery evaluation.

The *magic decorrelation* of Seshadri et al. [24] is an algorithmic approach to the unnesting of subqueries that extends the Starbust query optimizer [22]. Their approach is essentially a generalization of the principles of [6, 15], evaluating the subqueries using a join or left-outer join construct and temporary tables [24].

Galindo-Legaria and Joshi [14] solve subqueries in SQL Server by decomposing them into primitive orthogonal pieces (the *APPLY* operator), and then removing the correlations by transforming them into joins, semi-joins, outer-joins, setminus, etc., according to a set of rules built upon the same principles as earlier algorithms. Almost identical approaches were independently proposed by Celis and Zeller [7] and Wang et al. [26]

Unnesting has also been considered for object-oriented databases (e.g., [18, 13]). However, their results are not directly applicable in our context.

1.2 Contributions

With the exception of [23], all algorithms proposed for relational databases are based on rewriting the nested query expression into either some form of joins or set difference. It has previously been shown that it may not always be efficient to rewrite all subqueries to these forms [22, 23]. This is particularly true in an OLAP context with huge fact tables, where performing joins or set-difference operations can be extremely costly.

In this paper, we propose an alternative approach to evaluating subqueries based on counting. We note that although aggregation has been utilized in prior approaches, this is only done where the query already contains aggregation, whereas our approach uses counting as the central mechanism of the query rewriting. To the best of our knowledge, no previous research paper systematically formalizes and evaluates the use of counts to compute all forms of subqueries.

Our algorithm transforms nested subqueries into an algebra extended with the *GMDJ operator* for complex OLAP [10, 2]. The use of the GMDJ operator permits us to leverage OLAP optimizations, and thus enhances the space of solutions that can be considered by an OLAP query optimizer. Our algorithm is a stable, efficient, and flexible approach to evaluating subqueries.

1.3 Paper Outline

The paper proceeds as follows: Section 2 presents the algebra used in the paper, introduces the GMDJ operator, and presents our motivating example. Section 3 discusses the transformation from nested query expressions to GMDJ expressions and gives an algorithm. Section 4 discusses a couple of GMDJ optimizations that are particularly relevant to subquery expressions. Section 5 empirically compares the performance of GMDJ expressions with other subquery processing methods. Section 6 concludes the paper.

2 Preliminaries

2.1 Nested Query Expressions

In this paper, we express nested query expressions using an extended version of the nested query algebra proposed by Bækgaard and Mark [5], with the semantics of the nested query expressions defined to comply with the semantics of the SQL subquery constructs that they represent [25]. Let $\phi \in \{=, >, <, \leq, \geq, \neq\}$. Following this scheme, we subdivide the nested query constructs considered in the paper into the following types.

A *nested comparison selection* has the form $\sigma[x\ \phi\ S]B$, where S is a single-tuple, single-attribute algebraic expression. A *quantified nested comparison selection* has the form $\sigma[x\ \phi_{some}\ S]B$ or $\sigma[x\ \phi_{all}\ S]B$, where S is an algebraic expression. A *nested existential selection* has the form $\sigma[\exists S]B$ or $\sigma[\nexists S]B$, where S is an algebraic expressions.

The above algebraic constructs mirror the standard SQL subquery constructs. The ANY predicate is a synonym for SOME, and IN ($\in$) and NOT IN ($\notin$) are defined as follows [25]:

$$\sigma[x \in \pi[y](R)] \equiv \sigma[x\ =_{some}\ \pi[y](R)]$$
$$\sigma[x \notin \pi[y](R)] \equiv \sigma[x\ \neq_{all}\ \pi[y](R)]$$

A *free reference* [5], is an attribute reference in a predicate that does not refer to any relation within the local scope of its selection condition. A *correlation predicate* is a selection predicate that contains a free reference.

The nested algebra given above has the advantage that it directly maps to the subquery constructs of an SQL-like OLAP query language. We note that our transformation rules are not dependent on the use of this nested algebra; it would be equally easy to derive the transformations given below for other algebra variants as well; e.g., we could map to GMDJs from the *APPLY* operator (used to represent looping subquery evaluation) of [14] in the same way.

2.2 The Generalized Multi-Dimensional Join Operator

The *generalized multi-dimensional join* operator (in the rest of this paper referred to simply as the GMDJ) allows for the algebraic expression of many complex OLAP queries [10, 2]. A salient feature of the operator is that it provides a clean separation between the definition of the groups and the definition of the aggregate in the OLAP query, and thus allows the expression of a series of complex aggregations over different data partitions using a single algebraic operator.

Let B and R be relations, b a tuple in B and θ be a (complex) condition over B and R. We write $attr(\theta)$ to denote the set of attributes used in θ. $RNG(b,R,\theta) = \{r | r \in R \wedge \theta(b,r)\}$ denotes the range of b; i.e., the set of tuples in R that satisfies θ given b. We use $\{\{\ldots\}\}$ to denote a multiset.

Definition 2.1 Let $B(\mathbf{B})$ and $R(\mathbf{R})$ be relations, θ_i be a condition with $attr(\theta_i) \subseteq \mathbf{B} \cup \mathbf{R}$, and l_i be a list of aggregate functions $(f_{i1}, f_{i2}, \ldots, f_{in_i})$ over attributes $c_{i1}, c_{i2}, \ldots, c_{in_i}$ in $\mathbf{R}$. The GMDJ, $MD(B, R, (l_1, \ldots, l_m), (\theta_1, \ldots, \theta_m))$, yields a relation with schema[1]: $\mathbf{X} = (\mathbf{B}, f_{11}\_R\_c_{11}, \ldots, f_{1n_1}\_R\_c_{1n_1}, f_{m1}\_R\_c_{m1}, \ldots, f_{mn_m}\_R\_c_{mn_m})$ whose instance is determined as follows. Each tuple $b \in B$ contributes to an output tuple $\mathbf{x}$, such that:

- $\mathbf{x}[A] = b[A]$, for every attribute $A \in \mathbf{B}$
- $\mathbf{x}[f_{ij_i}\_R\_c_{ij_i}] = f_{ij_i}\{\{t[c_{ij_i}] | t \in RNG(b, R, \theta_j)\}\}$, for every attribute $f_{ij_i}\_R\_c_{ij_i}$ of $\mathbf{x}$.

□

The GMDJ is a composite operator that neatly summarizes a complex algebra expression. Like a join it takes two operands: B, the *base-values relation* and R, the *detail relation*. It returns a single output relation X, consisting of the tuples of B extended by the computed aggregate functions of $l_1, \ldots, l_m$. For each tuple $x \in X$, the aggregate functions in l_i only range over the tuples of R that fulfill θ_i. The GMDJ offers a great deal of flexibility when defining aggregation, as B and R can be arbitrary tables and θ can be an arbitrary predicate defined over B and R.

As demonstrated in prior papers (e.g., [11, 9]), the GMDJ can be evaluated very efficiently. In fact, the results of Rao and Ross [23] can be viewed as one of the many optimization schemes for the GMDJ evaluation. The results presented in this paper allow us to exploit the techniques of [23] to optimize the evaluation of all the subqueries, rather than just a small subset of them.

[1]Attributes are appropriately renamed if there are any duplicate names generated this way. In our examples we write $E \rightarrow C$ to rename E to C.

Note that [14] introduces a variant on the *apply* operator referred to as *SEGMENT-APPLY*, a primitive operator built on the concepts of segmented evaluation presented in [11]. However, because *SEGMENT-APPLY* is considered only as a special-case operator, the algorithm in [14] does not exploit the concept. Using the GMDJ, we demonstrate how segmented evaluation techniques can be generally extended to the evaluation of subqueries by using the power of counting.

2.3 Motivating Examples

Consider an IP provider wishing to analyze IP flows. An IP flow is a sequence of packets transferred from a given source to a given destination, using a given `Protocol`. All packets in a flow pass through a given router, which maintains summary statistics about the flow and dumps out a tuple for each flow passing through it. Assume a data warehouse with the following schema.

```
User (UserIP, Name, Address, Balance)
Flow (RouterId, SourceIP, SourcePort, DestIP,
      DestPort, Protocol, StartTime, EndTime,
      NumPackets, NumBytes)
Hours(HourDescription, StartInterval,
      EndInterval)
```

Hours is a time dimension used for specifying complex OLAP queries.

Example 2.1 Assume we want to ask the following: "On an hourly basis, what fraction of the traffic is due to web traffic?" This very simple question is actually a fairly complex OLAP query, requiring multiple joins and aggregations if we were to attempt to express it using standard algebraic operators. It can be expressed with a single GMDJ operator, as follows:

$$\pi[\texttt{HourDescription}, \texttt{sum1}/\texttt{sum2}]\sigma[cnt1 = cnt2]$$
$$(MD(Hours \rightarrow H, Flow \rightarrow F, (l_1, l_2), (\theta_1, \theta_2)))$$

where:

$l_1 : (sum(F.NumBytes) \rightarrow sum1)$
$l_2 : (sum(F.NumBytes) \rightarrow sum2)$
$\theta_1 : (F.StartTime >= H.StartInterval \wedge F.StartTime < H.EndInterval \wedge F.Protocol = "HTTP")$
$\theta_2 : (F.StartTime >= H.StartInterval \wedge F.StartTime < H.EndInterval)$

Figure 1 shows two input table (on the left) and the resulting output table (with the results $sum1$ and $sum2$ left unreduced) generated by the GMDJ above.
□

Because of the split between the "grouping" and aggregation aspect of the complex OLAP expression, we are able

`Hours`

HourDsc.	StartInterval	EndInterval
1	0	60
2	61	120
3	121	180

`Flow`

StartTime	Protocol	NumBytes	...
43	HTTP	12	...
86	HTTP	36	...
99	FTP	48	...
132	HTTP	24	...
156	HTTP	24	...
161	FTP	48	...

Result

HourDsc.	sum1/sum2
1	12/12
2	36/84
3	48/96

Figure 1 : Input table Hours and Flows and Output table for GMDJ in Example 2.1

identify common sub-expressions and express with a single GMDJ an operation that would require three joins and two aggregation operations using a conventional algebra. In cases where the base-values table (the Hours table in the above example) fits into main-memory, it would be possible to evaluate this query using GMDJs in *a single scan of the detail table*. Even in those cases where in-memory computation is not possible, simple memory management techniques allow us to avoid unnecessary buffer thrashing and compute the GMDJ at a well-defined cost. The ability to efficiently evaluate queries that would otherwise require multiple, expensive join operations is one of the salient features of the GMDJ operator.

Example 2.2 Let us extend the OLAP query given above, to examine the trends for specific hour intervals. We wish to find out: "For each hour in which there exists traffic to DestIP 167.167.167.0, what fraction of the total traffic is due to web traffic?" This is expressed as:

$$\pi[\texttt{HourDescription}, \texttt{sum1/sum2}]\sigma[cnt1 = cnt2] \\ (MD(B, Flow \rightarrow F_O, (l_1, l_2), (\theta_1, \theta_2)))$$

where:

$$B = \sigma[\exists\sigma[F_I.DestIP = 167.167.167.0 \wedge \\ F_I.StartTime >= H.StartInterval \wedge \\ F_I.StartTime < H.EndInterval] \\ (Flow \rightarrow F_I)]Hours \rightarrow H$$
$$l_1 : (sum(F_O.NumBytes) \rightarrow sum1)$$
$$l_2 : (sum(F_O.NumBytes) \rightarrow sum2)$$
$$\theta_1 : (F_O.StartTime >= B.StartInterval \wedge \\ F_O.StartTime < B.EndInterval \wedge \\ F_O.Protocol = "HTTP")$$
$$\theta_2 : (F_O.StartTime >= B.StartInterval \wedge \\ F_O.StartTime < B.EndInterval)$$

□

The subquery construct B in Example 2.2 is easy to unnest using conventional unnesting algorithms as it contains a simple `EXISTS` subquery that can be converted to a join or semi-join. However, as we will demonstrate in Section 3, it can equally easily be converted to a GMDJ expression, which allows us to leverage a variety of powerful optimizations that can result in improved performance.

Example 2.3 Assume we want to compute the total amount of traffic originating at an IP address and the total amount of traffic received by this IP address. We only want to do this for SourceIPs for which there does not exist any flows to DestIP 167.167.167.0, there exist flows to DestIP 168.168.168.0, and there do not exist any flows to DestIP 169.169.169.0. The above query is very easy to express using subqueries, as follows:

$$B = \sigma[\nexists\sigma[\theta_{S1}](Flow \rightarrow F_1) \wedge \exists\sigma[\theta_{S2}](Flow \rightarrow F_2) \wedge \\ \exists\sigma[\theta_{S3}](Flow \rightarrow F_3)]\pi[SourceIP](Flow \rightarrow F_O)$$

where:

$$\theta_{S1} : (F_0.SourceIP = F_1.SourceIP \wedge \\ F_1.DestIP = "167.167.167.0")$$
$$\theta_{S2} : (F_0.SourceIP = F_2.SourceIP \wedge \\ F_2.DestIP = "168.168.168.0")$$
$$\theta_{S3} : (F_0.SourceIP = F_3.SourceIP \wedge \\ F_3.DestIP = "169.169.169.0")$$

The aggregate can still be computed using GMD-joins:

$$\pi[\texttt{SourceIP}, \texttt{sumTo}, \texttt{sumFrom}] \\ (MD(B, Flow \rightarrow F, (l_1, l_2), (\theta_1, \theta_2)))$$

where:

$$l_1 : (sum(F.NumBytes) \rightarrow sumTo)$$
$$l_2 : (sum(F.NumBytes) \rightarrow sumFrom)$$
$$\theta_1 : (B.SourceIP = F.SourceIP)$$
$$\theta_2 : (B.SourceIP = F.DestIP)$$

□

Because of the separation of grouping and aggregation in the GMDJ, it is easy to define this rather complex OLAP query using subqueries. The base-values table is a simple distinct projection with multiple `EXISTS` subqueries. Such subqueries can be relatively easily unnested; e.g., to joins/outer-joins using conventional unnesting algorithms. However, performing self-joins on the fact table of a data warehouse (which is what unnesting using joins in the above case would result in) is however not necessarily the best idea, and even heavily optimized, the join expressions derived for the above queries are likely to be extremely costly. In the following, we will demonstrate how the flexibility of the GMDJ operator allow us to evaluate the aggregates of the complex query involving three subqueries and a complex aggregation in just *a single scan of the* `Flow` *table*.

Nested Query Expression	GMDJ expression
$\sigma[B.x\ \phi\ \pi[R.y]\sigma[\theta](R)]B$	$\sigma[cnt = 1](MD(B, R, (count(*) \rightarrow cnt), \theta \wedge B.x\ \phi\ R.y))$
$\sigma[B.x\ \phi\ \pi[f(R.y)]\sigma[\theta](R)]B$	$\sigma[x\ \phi\ fy](MD(B, R, (f(y) \rightarrow fy), \theta))$
$\sigma[B.x\ \phi_{some}\ \pi[R.y]\sigma[\theta](R)]B$	$\sigma[cnt > 0](MD(B, R, (count(*) \rightarrow cnt), \theta \wedge R.x\ \phi\ B.y))$
$\sigma[B.x\ \phi_{all}\ \pi[R.y]\sigma[\theta](R)]B$	$\sigma[cnt1 = cnt2](MD(B, R, ((count(*) \rightarrow cnt1), (count(*) \rightarrow cnt2)),$ $((\theta \wedge R.x\ \phi\ B.y), (\theta))))$
$\sigma[\exists\sigma[\theta](R)]B$	$\sigma[cnt > 0](MD(B, R, (count(*) \rightarrow cnt), \theta))$
$\sigma[\nexists\sigma[\theta](R)]B$	$\pi[A]\sigma[cnt = 0](MD(B, R, (count(*) \rightarrow cnt), \theta))$

Table 1. Mapping Nested Query Expressions into GMDJs

3 Computing Subqueries using GMDJs

In this section, we develop an algorithm to efficiently evaluate subqueries expressed with the nested algebra given in Section 2.1, using GMDJs. We first discuss how to map a simple nested query expression with a single subquery into a GMDJ expression. We extend this basic algorithm to discuss how multiply nested subqueries are handled using our basic approach. Section 3.3 presents the full algorithm.

3.1 SQL Query with a Single Subquery

Let B and R be tables. Then we translate a query expression Q with a single nested query expression S as follows.

Theorem 3.1 *Let Q be an algebraic query expression $\sigma[S](B)$, where S is a nested query expression. Let R be a table, x and y be attributes, $f(y)$ be an aggregate function over the attribute y, and θ be the correlation condition of subquery S. Let $\phi \in \{=, >, <, \leq, \geq, \neq\}$, and let $\rightarrow$ denote the renaming of an attribute. Table 1 maps the query Q with nested query expressions to an equivalent GMDJ expression $\sigma[C](MD(B, R, l, \theta'))$.*

Proof: Let $X = \sigma[\psi]MD(B \cup \{b\}, R, l, \theta')$, where ψ and θ' are the selection and aggregate-conditions specified in Theorem 3.1. We demonstrate that tuple b is in X iff tuple b is in $\psi = \sigma[S](B \cup \{b\})$. Observe that within the limits of the query form defined in Theorem 3.1, it is sufficient for the proper handling of `NULL` values if the algebraic expression selects b if the subquery predicate returns `true`. This is so because a tuple evaluating to `false` or unknown is discarded—the so-called *where-clause truncation* [21].

The **comparison predicate** over an attribute returns unknown if either of the operands is the `NULL` value, otherwise returning `true` if $b.x\ \phi\ R.x$. Since unknown = `false` in this context, it follows that Q will contain the tuple b iff $b.x\ \phi\ R.c$ and $\theta(b, R.c)$ is `true`. This is equivalent to saying that $RNG(R, b, (b.x\ \phi\ R.y \wedge \theta))$ must contain one tuple. The `count(*)` of our tuple b in the GMDJ translation for the comparison predicate returns precisely $|RNG(b, R, (b.x \phi R.y \wedge \theta))|$ (this follows from Definition 2.1). Thus, $\sigma[cnt = 1]$ selects the tuple b iff $RNG(R, b, (b.x\ \phi\ R.y \wedge \theta))$ returns exactly one tuple. We note that the comparison predicate is only well-defined if the inner block returns only one tuple, otherwise generating a run-time exception. Handling such run-time exceptions is beyond the scope of this paper.

The **comparison predicate** over an aggregate function follows the same 3VL rule as above, returning `true` if $b.x\ \ \phi\ \ f(R.y)$. It follows that Q will contain the tuple b iff $b.x\ \ \phi\ \ f(R.y)$, where $f(R.y) = f\{\{R.y|r \in RNG(b, R, \theta)\}\}$. Following Definition 2.1, we find that the GMDJ translation computes $f(R.y)$ for $RNG(b, R, \theta)$, and then performs the selection $b.x\ \ \phi\ \ f(R.y)$. Thus the algebraic expression with the GMDJ is a direct mapping of the definition of the subquery construct [25].

The **quantified comparison predicate** `some` is `true` if the comparison predicate evaluates to `true` for at least one tuple in R, `false` if S is empty or if the comparison predicate evaluates to `false` for all tuples in R, unknown otherwise. It follows that b is in Q iff $RNG(b, R, (b.x\ \phi\ r.y \wedge \theta))$ is greater than 0 (anything else results in `false` or unknown). The `count(*)` of our tuple b in the GMDJ translation for the quantified comparison predicate `some` returns precisely $|RNG(b, R, (b.x\ \phi\ R.y \wedge \theta))|$. Thus $\sigma[cnt > 0]$ returns the tuple b in its result iff $\exists r \in R|(b.x\ \ \phi\ \ r.y \wedge \theta(b, r))$.

The **quantified comparison predicate** `all` is `true` if R is empty or if the comparison predicate evaluates to `true` for every tuple in R, `false` if the comparison predicate evaluates to `false` for at least one tuple in R, and unknown otherwise. If R is empty, then $cnt1$ and $cnt2$ both evaluate to 0 and b is selected.[2] If R is not empty, then the query Q contains the tuple b iff $\forall r \in RNG(b, R, \theta)|b.x\ \ \phi\ \ r.y$. This corresponds to stating that $RNG(b, R, \theta) \dot{-} RNG(b, R, (\theta \wedge b.x\ \ \phi\ \ r.y)) =$

[2]The commonly held assumption sometimes seen in the literature that an `all` predicate can be reduced to a subquery using aggregate functions (e.g., $B.x >$ `all` $R.y$ is proposed equivalent to $B.x >$ `max`$(R.y)$) is incorrect. While seemingly a reasonable assumption, it breaks down when b correlates to the empty table; while the `all` predicate will return `true`, the aggregate query will return unknown(`max` of nothing evaluates to `NULL`).

$\emptyset$. The GMDJ expression for `all` computes two counts: $|RNG(b,R,(\theta \wedge b.x\ \phi\ r.y))|$ and $|RNG(b,R,\theta)|$. Since $RNG(b,R,(\theta \wedge b.x\ \phi\ r.y)) \subseteq RNG(b,R,\theta)$, it follows that if $|RNG(b,R,(\theta \wedge b.x\ \phi\ r.y))| = |RNG(b,R,\theta)|$, then $RNG(b,R,(\theta \wedge b.x\ \phi\ r.y)) = RNG(b,R,\theta)$.

The `exists` **predicate** $\exists$ returns the tuple b iff $|RNG(b,R,\theta)| > 0$ [25], otherwise returning `false`. The GMDJ expression directly maps this definition.

The `not exists` **predicate** $\nexists$ returns the tuple b iff $|RNG(b,R,\theta)| = 0$ [25], otherwise returning `false`. The GMDJ expression directly maps this definition. □

The key observation in the mapping given in Table 1 is that we transform a selection over a complex predicate (the nested query expression) into a simply selection over a complex aggregate expression (the GMDJ). Note the systematic use of counting for the subquery evaluation. As demonstrated in [10, 2, 1], GMDJs permit an efficient evaluation of the complex aggregate expression.

We note that the resulting GMDJ expressions are regular algebraic expression and **not** nested query expressions. The GMDJ is a binary operator that takes two tables as input just like a join; an identical notation for joins would write the expression $R_1 \bowtie_\theta R_2 \bowtie_\theta R_3$ as $\bowtie (R_3, (\bowtie (R_1, R_2, \theta), \theta)$. An alternate join-like notation for the GMDJ operator, would be to write the GMDJ $MD(B,R,l,\theta)$ on the following form: $B^{MD} \bowtie^l_\theta R)$. Like joins, the GMDJ can commute with other algebraic operators (projections, selections, joins, etc.) under the appropriate conditions.

Example 3.1 Consider the nested query expression in Example 2.2. Following the transformation rules of Table 1, we can rewrite the query with nested expressions into the following form:

$$\pi[\texttt{HourDescription}, \texttt{sum1/sum2}]$$
$$(MD(B, Flow \to F_O, (l_1,l_2),(\theta_1,\theta_2)))$$

where $l_1, l_2, \theta_1, \theta_2$ are defined as in Example 2.2, and:

$$B = \sigma[cnt > 0] MD(Hours \to H, Flow \to F_I, l_S, \theta_S)$$
$$l_S : (count(*) \to cnt)$$
$$\theta_S : F_I.DestIP = 167.167.167.0 \wedge F_I.StartTime >= H.StartInterval \wedge F_I.StartTime < H.EndInterval$$

□

Theorem 3.1 can be used to translate any query with a single nested expression block into an equivalent algebraic expression using GMDJs. Occasionally, however, queries will contain more than one subquery block. Such subquery blocks can be either nested at the same level, or a subquery blocks nested within another subquery block.

Subquery blocks nested at the same level are handled quite easily by computing all of the aggregates required first, and then performing the selections (as given by Theorem 3.1) corresponding to the subquery predicates.

Example 3.2 Consider the nested expression in the OLAP query of Example 2.3, which contains multiple subquery expressions. This can be rewritten into the following algebraic expression containing GMDJs (with $l_1, l_2, \theta_1, \theta_2$ defined as for Example 2.3).

$$\pi[\texttt{SourceIP}, \texttt{sumTo}, \texttt{sumFrom}]$$
$$(MD(B, Flow \to F, (l_1,l_2),(\theta_1,\theta_2)))$$

where:

$$B = \sigma[cnt1 = 0 \wedge cnt2 > 0 \wedge cnt3 = 0]$$
$$MD(MD(MD((\pi[SourceIP]Flow \to F_0), (Flow \to F_1), l_{S1}, \theta_{S1}), (Flow \to F_2), l_{S2}, \theta_{S2}), (Flow \to F_3), l_{S3}, \theta_{S3})$$
$$l_{S1} : (count(*) \to cnt1)$$
$$l_{S2} : (count(*) \to cnt2)$$
$$l_{S3} : (count(*) \to cnt3)$$
$$\theta_{S1} : (F_0.SourceIP = F_1.SourceIP \wedge F_1.DestIP = \text{"167.167.167.0"})$$
$$\theta_{S2} : (F_0.SourceIP = F_2.SourceIP \wedge F_2.DestIP = \text{"168.168.168.0"})$$
$$\theta_{S3} : (F_0.SourceIP = F_3.SourceIP \wedge F_3.DestIP = \text{"169.169.169.0"})$$

Since GMDJs can commute with each other, the ordering of the GMDJs in B can be arbitrary. As we will demonstrate in Section 4, the flexibility of the GMDJ will allow us to significantly optimize this expression using simple algebraic transformations. □

3.2 Linearly-nested Subqueries

Linearly-nested subqueries are query expressions where the nested expressions contains additional nested query expressions. We distinguish between two forms of correlation predicates in this case: *neighboring predicates* reference only tables in their own query expression and tables of the immediately enclosing query expression. All other correlation predicates are *non-neighboring predicates* [20].

Theorem 3.2 *Let B be a table, and $\theta_1, \ldots, \theta_n$ be the correlation predicates over B and $R_1, \ldots, R_n$ respectively. Let $\phi \in \{=,>,<,\leq,\geq,\neq\}$, and ψ_i be a predicate such that: $\psi_i := x\ \phi_i \mid x\ \phi_{i,\texttt{some}} \mid x\ \phi_{i,\texttt{all}} \mid \exists \mid \nexists$. Let $\varpi \in \{\vee, \wedge\}$. Let C_i, l_i, and θ'_i be the appropriate conditions derived using Theorem 3.1 for subquery S_i. Then a linear nested query expression on the following form:*

$$\sigma[\psi_2(\sigma[\theta_2\ \varpi\ \psi_1 \sigma[\theta_1](R_1)](R_2))](B)$$

Can be transformed into the following GMDJ expression:

$$\sigma[\upsilon_2] MD(B, (MD(R_2, R_1, l_1, \theta'_1)), l_2, \theta'_2\ \varpi\ \upsilon_1)$$

Theorem 3.2 extends Theorem 3.1 to multi-block queries. Intuitively, we simply transform the inner-most nested expression to GMDJs, prior to transforming the outer nested expression. Thus the inner-most nested query block

(R_1), forms the detail table for a GMDJ expression with R_2 as its base-values table. This GMDJ expression then forms the detail table for an additional GMDJ expression.

Theorem 3.2 is an intuitive extension of the basic theorem. However, it needs to be extended further if it is to handle non-neighboring correlation predicates. To illustrate the problem, consider the following example.

Example 3.3 We want to know the user accounts that have been active (i.e., have been the source of traffic) in each hour since January 1, 2002. Such a query is easiest to express through a double existential negation; essentially, there is no hourly interval for which there has not been traffic from the user's IP. This can be expressed as:

$$\sigma[\nexists(\sigma[\theta_H \wedge (\nexists\sigma[\theta_F](Flow \rightarrow F))](Hours \rightarrow H))]\\(User \rightarrow U)$$

where:

$$\theta_H : H.StartInterval > \text{"}00:00:0001-01-02\text{"}$$
$$\theta_F : F.StartTime >= H.StartInterval \wedge F.StartTime\\ < H.EndInterval \wedge F.SourceIP = U.IPAddress$$

Transforming the query to remove the nested query expressions, we derive the following:

$$\sigma[cntH = 0\wedge]MD(User \rightarrow U, (\sigma[cntF = 0]\\ MD(Hours \rightarrow H, Flow \rightarrow F, l_F, \theta_F)), l_H, \theta_H)$$

where:

$$l_H : (count(*) \rightarrow cntH)$$
$$l_F : (count(*) \rightarrow cntF)$$

As we can see, θ_F contains the non-neighboring predicate $F.SourceIP = User.IPAddress$, which leads to a θ-condition in the GMDJ with a reference to the table `Flow`. This is a clear violation of the the semantics of the GMDJ, which requires $attr(\theta) \subseteq \mathbf{B} \cup \mathbf{R}$. □

To safely handle non-neighboring predicates, we can push down the source table of the outer query to our detail tables. The following two transformation rules of the GMDJ, permit us to handle non-neighboring predicates within the GMDJ framework.

Theorem 3.3 *Let B and R be tables, l a list of aggregate functions, and θ a condition. Then:*

$$MD(B, R, l, \theta) = MD(B, (B \ltimes_\theta R), l, \theta)$$

Theorem 3.4 *Let B, R, and T be tables, l a list of aggregate functions, C a join condition and θ the condition of the GMDJ. Then:*

$$T \ltimes_C MD(B, R, l, \theta) = MD((T \ltimes_C B), R, l, \theta)$$

The proofs of the above follow from the semantics of the GMDJ (see [1] for the details). These basic transformation rules allows us to push down the outer-most base-values table until all non-neighboring correlation predicates in the GMDJ expression are legal.

We note that the only case where there is a need to introduce joins into the algorithm is in the case of non-neighboring predicates. Essentially, these expressions will result in expressions similar to join/outer-join unnesting methods [12, 20], with GMDJs in place of their aggregation operators. Optimized correctly to take advantage of similar joins that may be generated by multiple push downs, the GMDJ algorithm will require the computation of $n-1$ joins (where n is equivalent to the depth of the non-neighboring predicate); the exact same number as would be required if we employed a conventional join/outer-join strategy.

In general, non-neighboring predicates do not have a nice solution [20, 4], and require the use of supplementary joins. However, the GMDJ algorithm still performs well compared to conventional unnesting. Consider, for example, the query of Example 3.3.

Example 3.4 Following Propositions 3.3 and 3.4 the translation of the nested query in Example 3.3 yields the following algebraic expression:

$$\sigma[cntH = 0]MD(User \rightarrow U, (\sigma[cntF = 0]\\ MD((User \ltimes_{\theta_H} Hours) \rightarrow H, Flow \rightarrow F, l_F, \theta_F)),\\ l_H, \theta_H)$$

Thus we find that we can handle this nested query expression with non-neighboring predicates by adding a single join to the query expression.

For comparison, consider the same query unnested using the *APPLY* operator used in Microsoft SQL Server 8. Replacing the nested queries with *APPLY* operators and then removing these, we would derive the following query expression [14]:

$$User \setminus_{A_U} ((User \ltimes_{\theta_H} Hours) \setminus_{A_{UH}} (User \ltimes_{\theta_{FU}}\\ (Hours \ltimes_{\theta_{FH}} Flow)))$$

where:

$$\theta_H : H.StartInterval > \text{"}00:00:0001-01-02\text{"}$$
$$\theta_{FH} : F.StartTime >= H.StartInterval \wedge\\ F.StartTime < H.EndInterval$$
$$\theta_{FU} : F.SourceIP = User.IPAddress$$
$$A_U : Schema\ of\ \texttt{User}$$
$$A_{UH} : Schema\ of\ \texttt{User} \bowtie \texttt{Hours}$$

Essentially, the resulting query is the definition of relational division; which expresses the universal quantification query of Example 3.3. It should be obvious from this example that, even for nested query expressions with non-neighboring predicates, the leveraging of GMDJ expressions can potentially result in significant improvements in query evaluation. □

```
Algorithm SubqueryToGMDJ(Q)

    Apply de Morgan's laws to push down negations to the atomic predicates
    Eliminate negations in front of subqueries using the rules:
        $\neg(t\ \phi\ S) \Longrightarrow t\ \bar{\phi}\ S$, $\neg(t\ \phi_{some}S) \Longrightarrow (t\ \bar{\phi}_{all}S)$, $\neg(t\ \phi_{all}S) \Longrightarrow (t\ \bar{\phi}_{some}S)$
    Let $X = \pi[A]\sigma[W]MD(B, \emptyset, \{\{\}\}, \mathtt{true})$
    Iterate over $X$ repeatedly until all subqueries $S_i$ are eliminated {
        $\sigma[W_i]MD(B,R,l,\theta) \Rightarrow \sigma[W_i^!]MD((MD(B,R,l_n,\theta)),R_i,l_i,\theta_i')$
        $\sigma[W]MD(B,R,l,S_i \varpi \theta) \Rightarrow \sigma[W]MD(B,(MD(R,R_i,l_i,\theta_i)),l,C_i \varpi \theta)$
    }
    Push down tables to non-neighbor predicates as appropriate (cf. Propositions 3.3 and 3.4)
    return $X$
```

3.3 The Integrated Algorithm

This section synthesizes the theorems into an integrated algorithm that translates general nested query expression into an algebraic expression with GMDJs. Pushing down and eliminating negations in the subquery expressions ensures that NULL values in the data are handled correctly [4].

Theorem 3.5 *Let Q be a query expression with nested predicates on the following form $\sigma[W]B$, where W is a predicate $W ::= \neg(W) \mid W \wedge W \mid W \vee W \mid P$ and P is a comparison predicate or subquery expression S_i. Let S_i be on the form of Q, and let C_i, l_i, and θ_i' be the selection, aggregate functions, and theta conditions as determined by Theorem 3.1 for the subquery predicate S_i. R_i is the source table of subquery S_i. Let $\emptyset$ be the empty table and $\varpi \in \{\wedge, \vee\}$. Let W_i represent a general predicate containing S_i, then $W_i^!$ represents the same predicate with the condition S_i replaced by C_i. The algebraic expression returned by Algorithm* SubqueryToGMDJ(Q) *correctly computes Q.*

Algorithm SubqueryToGMDJ is a simple and efficient algorithm that correctly translates a nested query expression containing one or more arbitrarily nested subqueries, to a GMDJ expression.

4 GMDJ Optimizations for Subqueries

GMDJs can be evaluated very effectively, and both analytical and experimental studies of the GMDJ expressions constructed by Algorithm SubqueryToGMDJ indicate that this basic algorithm provides an efficient way to evaluate subqueries. However, introduction of the GMDJ allows the leveraging of additional algebraic transformations and optimization techniques that are relevant in an OLAP context. A general treatment of the algebraic transformations possible is beyond the scope of this paper (see [10, 2, 1] for details); here we present only two general GMDJ optimizations that are well-suited for optimizing GMDJ expressions representing subqueries. We observe that many specialized subquery optimizations (e.g., early termination of a computation) are orthogonal to GMDJ query evaluation.

4.1 Coalescing of GMDJs

One of the salient properties of the GMDJ is that a sequence of GMDJs can be coalesced to form a single GMDJ, provided that both GMDJs in a sequence are over the same underlying base table and the GMDJs conditions are independent of each other. We note that the latter requirement is always true for nested tree subqueries. Thus, we only need to verify that the underlying tables of the conjunctive tree queries are identical to determine whether coalescing is possible.

Proposition 4.1 *Let B be a table, and S_1 to S_n be subquery expressions over tables R_1 to R_n. Let $\varpi \in \{\wedge, \vee\}$. If $R = R_1 = R_2 = \ldots = R_n$, then:*

$$\sigma[S_1\ \varpi_1\ S_2\ \varpi_2\ \cdots\ \varpi_1\ S_n](B)$$

can be transformed into the following GMDJ expression:
$\pi[A]\sigma[C_1\ \varpi_1\ C_2\ \varpi_2\ \cdots\ \varpi_{n-1}\ C_n]$
$MD(B, R, (l_1, l_2, \ldots, l_n), (\theta_1', \theta_2', \ldots, \theta_n'))$ □

Proposition 4.1 follows directly from the theorems for coalescing GMDJs [10, 2].

Example 4.1 Recall the query of Example 2.3. Since all three subqueries range over the same subquery table, it follows that we can apply Proposition 4.1. Thus, instead of generating three GMDJs to compute the three nested query expressions (as in Example 3.2), we can do with just a single GMDJ in the base-values table.

$$\pi[\mathtt{SourceIP}, \mathtt{sumTo}, \mathtt{sumFrom}] \\ (MD(B, Flow \to F, (l_1, l_2), (\theta_1, \theta_2)))$$

where $l_{S1}, l_{S2}, l_{S3}, l_1, l_2, \theta_1, \theta_2$ are defined as in Example 3.2, and:

$$
\begin{aligned}
&B = \sigma[cnt1 = 0 \wedge cnt2 > 0 \wedge cnt3 = 0] \\
&\quad MD((\pi[SourceIP]Flow \rightarrow F_O), (Flow \rightarrow F_S), \\
&\qquad (l_{S1}, l_{S2}, l_{S3}), (\theta_{S1}, \theta_{S2}, \theta_{S3})) \\
&\theta_{S1} : (\mathtt{F_O.SourceIP} = \mathtt{F_S.SourceIP}\ \& \\
&\quad \mathtt{F_S.DestIP} = \text{"167.167.167.0"}) \\
&\theta_{S2} : (\mathtt{F_O.SourceIP} = \mathtt{F_S.SourceIP}\ \& \\
&\quad \mathtt{F_S.DestIP} = \text{"168.168.168.0"}) \\
&\theta_{S3} : (\mathtt{F_O.SourceIP} = \mathtt{F_S.SourceIP}\ \& \\
&\quad \mathtt{F_S.DestIP} = \text{"169.169.169.0"})
\end{aligned}
$$

In this particular case, we could in fact improve upon the expression further by pushing up the selections and coalescing the GMDJs. This would result in the following:

$$
\begin{aligned}
&\pi[\mathtt{SourceIP}, \mathtt{sumTo}, \mathtt{sumFrom}] \\
&\quad \sigma[cnt1 = 0 \wedge cnt2 > 0 \wedge cnt3 = 0] \\
&\qquad (MD(((\pi[SourceIP]Flow) \rightarrow B, (Flow) \rightarrow F, \\
&\qquad (l_{S1}, l_{S2}, l_{S3}, l_1, l_2), (\theta_{S1}, \theta_{S2}, \theta_{S3}, \theta_1, \theta_2)))
\end{aligned}
$$

where $l_{S1}, l_{S2}, l_{S3}, l_1, l_2, \theta_1, \theta_2$ are defined as in Example 3.2, and:

$$
\begin{aligned}
&\theta_{S1} : (\mathtt{B.SourceIP} = \mathtt{F.SourceIP}\ \& \\
&\quad \mathtt{F.DestIP} = \text{"167.167.167.0"}) \\
&\theta_{S2} : (\mathtt{B.SourceIP} = \mathtt{F.SourceIP}\ \& \\
&\quad \mathtt{F.DestIP} = \text{"168.168.168.0"}) \\
&\theta_{S3} : (\mathtt{B.SourceIP} = \mathtt{F.SourceIP}\ \& \\
&\quad \mathtt{F.DestIP} = \text{"169.169.169.0"})
\end{aligned}
$$

We thus find that, given the base-values table $\pi[SourceIP]Flow$, a single scan of the `Flow` table suffices to compute all the aggregates required. □

Essentially, coalescing allows us to evaluate a complex expression representing *multiple* subqueries over the same table, in just a single scan of that table. This form of complex relation sharing is **not** easy for a conventional optimizer using joins and traditional aggregate operators to detect [11, 9]. Coalescing is a particularly useful optimization when evaluating queries over large amounts of data as it significantly reduces disk I/O; thus leveraging these forms of techniques is particularly important in an OLAP context.

4.2 Base-tuple Completion

Another GMDJ optimization that is particularly well-suited for the expressions generated by GMDJ queries is that of removing completed base tuples from the evaluation. During query evaluation, there will be some point when a tuple $b \in B$ is completed, i.e., when no further tuples yet to be evaluated in R can have any effect on the result of tuple b. If a tuple b is completed with respect to all conditions $\theta_i \in \{\theta_1, \ldots, \theta_m\}$, then we need not consider it during further computation of that GMDJ. This allows us to reduce the in-memory size of the base-result structure (e.g., by transferring the completed tuples to disk), and thus speed up the query processing.

Definition 4.1 An entry $x \in X$ of a GMDJ is said to be *completed* with respect to the GMDJ iff no tuples further into the query evaluation will affect the output of X with respect to x. An entry is *active* iff it is not completed. □

Recall the definition of the range function: $RNG(b, R, \theta) = \{r \in R | \theta(b, r)\ is\ true\}$. Let b_i be the i'th tuple of B and r_j be the j'th tuple of R. Conceptually then, the GMDJ evaluation algorithm computes $RNG(b, r, \theta_1), \cdots, RNG(b, r, \theta_m)$, $\forall b_i \in B, \forall r_j \in R$. We can use the concept of RNG to formalize our tuple completion criteria.

Theorem 4.1 *Let the expression* $Q = \pi[A]\sigma[|RNG| > 0](MD(B, R, (l_1, \ldots, l_m), (\theta_1, \ldots, \theta_m)))$. *Let* $b_i \in B$ *and* $r_j \in R$. *If* $A \cap (l_1 \cup \ldots \cup l_m) = \emptyset$ *and* $\theta_1(b_i, r_j) \wedge \ldots \wedge \theta_m(b_i, r_j)$ *is true, at any time during the evaluation of Q, then* $|RNG| > 0$ *and* b_i *is completed with respect to* Q.

Theorem 4.2 *Let the expression* $Q = \sigma[|RNG| = 0](MD(B, R, (l_1, \ldots, l_m), (\theta_1, \ldots, \theta_m)))$. *Let* $b_i \in B$ *and* $r_j \in R$. *If* $\theta_1(b_i, r_j) \wedge \ldots \wedge \theta_m(b_i, r_j)$ *is true, at any time during the evaluation of Q, then* $|RNG| <> 0$ *and* b_i *is completed with respect to* Q.

Essentially, Theorem 4.1 catches those tuples that will return true for the selection condition. The crucial point in this theorem is that the projection negates the need for a precise calculation of the aggregates in $l_1, \ldots, l_m$. Theorem 4.2, on the other hand, catches those tuples that will return false for the selection condition. GMDJ expressions such as the above rarely occur in standard, non-nested query expression. This form is common, however, in expressions derived using Algorithm `SubqueryToGMDJ`.

Example 4.2 Consider the GMDJ expression of Example 4.1. Observe that the range of the selection condition $(cnt1 = 0) \wedge (cnt2 > 0) \wedge (cnt3 = 0)$ is greater than 0 iff $cnt1 = 0$, $cnt2 > 0$, and $cnt3 = 0$. However it is 0 if either $cnt1 > 0$, $cnt2 = 0$, or $cnt > 0$. This implies that $|RNG| = 0$, iff $|RNG(b, Flow, \theta_{S1})| > 0 \vee |RNG(b, Flow, \theta_{S2})| = 0 \vee |RNG(b, Flow, \theta_{S3})| > 0$. It follows that if θ_{S1} or θ_{S3} evaluate to true for a tuple $r \in Flow$, then the tuple $b \in Flow$ can be discarded from the query result. □

In the general case, detecting completeness of tuples is undecidable. However, nested query expressions will tend to result in algebraic queries in which the optimization of Theorems 4.1 and 4.2 can easily be identified and used with advantage.

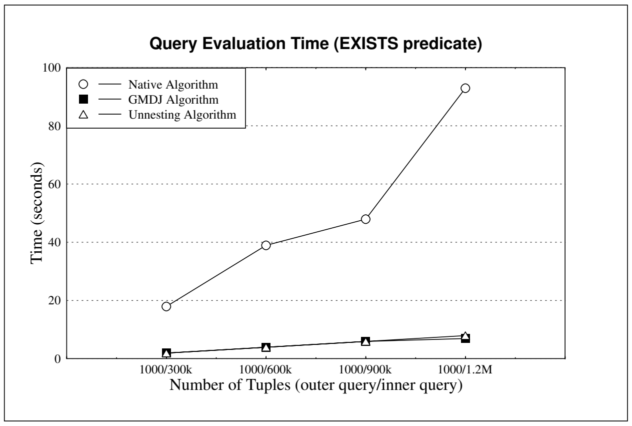

Figure 2 : `EXISTS` subquery

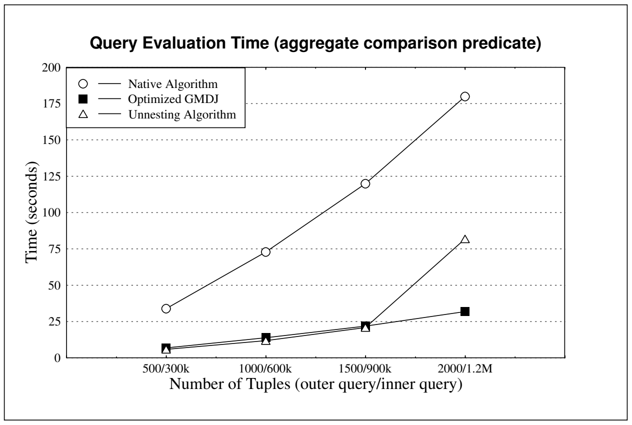

Figure 3 : Comparison predicate subquery

5 Experiments

In this section, we discuss a set of experiments comparing the performance of a GMDJ query engine computing nested query expressions with the performance of a commercial DBMS engine computing the query using its "native" mode, and a join/outer-join unnested translation of the nested query expressions. We derived four test databases from the TPC(R) `dbgen` program of between 50-200 Mbytes.

For the GMDJ processing, we used a primitive implementation of a GMDJ query engine that interfaces to the target DBMS through an API to fetch tuples and then processes the GMDJ using C++ code. The GMDJ query engine was limited to a hash index strategy and simple optimizations. We note that implementing the GMDJ processing outside of the DBMS results in a significant communications overhead between the GMDJ program and the DBMS; we believe that this cost cancels out (or may even exceed) any benefit gained by the external processing. Despite this, the implementation is superior to alternatives such as using conditional aggregations (e.g., `CASE` statements), due to the indexing mechanism intrinsic to GMDJ evaluation [2]. Prior experiments indicate that an integrated GMDJ engine would result in substantially better performance [9].

In the majority of our experiments, the initial performance of the target DBMS was typically 20-30% worse in its first attempt at query evaluation compared to subsequent runs; this is likely because the DBMS makes use of caching to speed up subsequent query runs. The average query times given in our graphs for nested loop and unnested join algorithms do not include the measurements for the first query run. We note that for all our test queries, the join/outer-joins (unnested) versions of the queries were translated using the best of the existing unnesting techniques developed in the literature for the particular query. All important attributes were indexed in the experiments, except when explicitly dropped to study the stability of the algorithms.

We first consider a nested query expression with an `EXISTS` subquery. The outer query block ranges over 1000 rows and the subquery block ranges over 300k, 600k, 900k, and 1.2M rows. The experimental results for the four sizes are given in Figure 2. It is worth noting that the native algorithm does not perform a nested loop; instead it employs a specialized algorithm for handling the `EXISTS` predicate. Despite this, both the join-based unnesting and the GMDJ evaluation perform significantly better. It is interesting to note that even for this type of query which is the simplest possible case in unnesting, the GMDJ performs just as well as joins.

Figure 3 gives the performance for a query containing comparison predicates with aggregate functions. The figure shows the performance for a query where the size of the outer query ranges from 500 to 2000 rows, and the inner block ranges from 300k to 1.2M rows. The native evaluation performs a simple nested loop. Not surprisingly, the join and GMDJ evaluations perform significantly better for this query. The performance of join-based evaluation degrades for the 1.2M subquery; despite using a sort-merge join, the optimizer seemed unable to handle the aggregate/outer-join query required efficiently. We note that the GMDJ evaluation is much more memory efficient and does not encounter such problems.

Figure 4 shows the performance for a query containing the quantified comparison predicate `ALL`. The table sizes for both the inner and outer query was 40k, 80k, 120k, and 160k rows. In our example the correlation predicate was a $<>$ on two key attributes. This is a good example of a query that is not well handled by existing unnesting algorithms. For an even smaller version of the query with 20k row table sizes; the join/outer-join unnesting took more than 7 hours to compute the result. Interestingly, we find

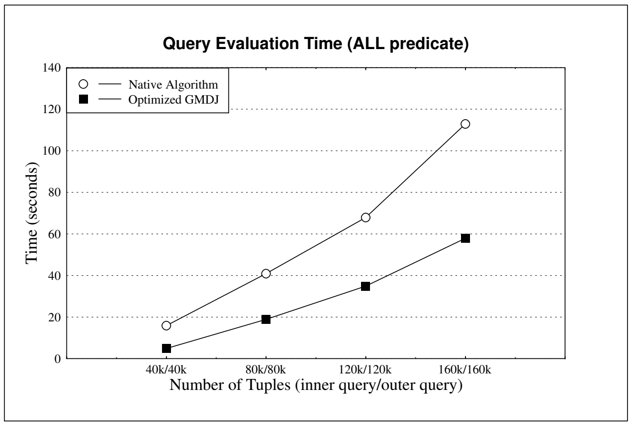

Figure 4 : Quantified comparison predicate `ALL`

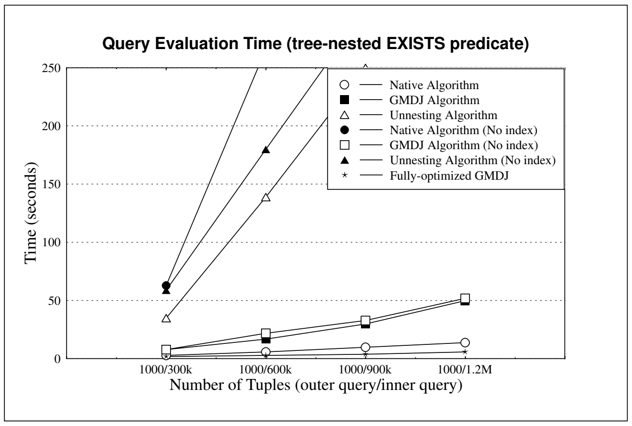

Figure 5 : Tree-nested `EXISTS` **predicates**

that the native evaluation performs very well for `ALL` subqueries. This is due to the target DBMS performing a *smart* nested loop algorithm, i.e., when encountering a nested tuple evaluating to `false` for the `ALL` condition, it discards the tuple *b* (essentially a form of tuple completion). Due to the semantics of the query, with the <> correlation predicate, the basic GMDJ evaluation algorithm is not particular effective (3 minutes for the 20k row version) and the GMDJ evaluation algorithm is forced into an evaluation that essentially mimics tuple-iteration semantics. However, if the GMDJ expressions are optimized using tuple completion, the GMDJs perform well.

Finally, Figure 5 shows the performance for a query containing two `EXISTS` subqueries ranging over 300k, 600k, 900k, and 1.2M row tables and nested within a query block containing 1000 rows. The correlation conditions of this query allow the native evaluation to perform very well on this query as long as the attributes used are indexed; where this is not the case it performs very badly. The join/outer-join version of this query needs to perform two large joins. Due to the disjoint predicates, it is impossible to combine the joins, and the DBMS struggles as a result of this. Without indexes, the join evaluation again performs very poorly.

As expected, the basic GMDJ unnesting algorithm performs very well, though not quite as well as the native algorithm. Because the GMDJ algorithm is not dependent on indexes on the source tables for the GMDJ evaluation itself, its performance is basically unaffected by the absence of indexes on the key attributes - in such a situation, the GMDJ evaluation performs an order of magnitude better. In addition, we find that by applying our previously described optimizations, the GMDJ evaluation again outperforms the specialized `EXISTS` evaluation used by our target DBMS.

Note that the optimized query performance above was achieved solely by the use of the simple optimizations given in Section 4. A cost-based optimizer deploying the full range of optimizations and techniques developed in the literature (see, e.g., [11, 23, 9, 10, 1]) should be capable of achieving significantly better performance.

6 Conclusions

In this paper we describe an approach to evaluating subqueries based on counting using the GMDJ operator. The performance of our algorithm matches that of conventional unnesting algorithms for simple subquery expressions, and can be seen to perform well even in situations where conventional unnesting performs poorly. This feature of our algorithm rests on a few simple facts:

- The algorithm uses GMDJs in place of joins, outer-joins, combination aggregation/joins, or set difference.
- Despite its complexity, the GMDJ operator can typically be evaluated in a single scan of the detail relation, or by an efficient algorithm that minimizes additional disk I/O [10, 1]; as this represents the dominant cost of the processing, this ensures that the GMDJ will perform as well or better than a join on identical relations.
- The algorithm only introduces additional joins when this is unavoidable (cf. Section 3.2). In all other cases, the algorithm avoid unnecessary supplementary tables and the number of algebraic operations are kept to the minimum.
- As the GMDJ operator can commute with projections, selections, joins, and other GMDJ operators, the GMDJ operator provides flexibility comparable to that of join reordering.

In the real world, ad-hoc queries may often lack useful indexes, leading to very expensive subquery computations for conventional unnesting algorithms (due to the unindexed joins). Note that none of the other unnesting algorithms reviewed are capable of maintaining stability of performance in the absence of indexes on the source table for all of the subqueries considered in this paper. The presence or absence of indexes on the base tables, however, has minimal or no effect on the GMDJ processing algorithm [2].

It has previously been noted that a wide variety of specialized OLAP evaluation optimization algorithms can be generalized as algebraic transformations using GMDJs [10]. It is interesting to observe that many complex subquery optimizations can similarly be generalized using GMDJs; e.g., the specialized "tuple completion" algorithm of our target DBMS and the reuse of invariants [23].

Because the GMDJ evaluation has a well-defined cost [1], it is easy to incorporate the GMDJ algorithm proposed in this paper into a cost-based framework. For example, one could introduce additional alternate correlation removal rules for the *APPLY* operator of [14] based on GMDJs, allowing the cost-based query optimizer to select between a rich set of alternatives (joins, set-division **and** GMDJs) for the subquery evaluation. We note that the GMDJ operator is well-suited to evaluation in a parallel or distributed DBMS environment [3]. Thus the GMDJ algorithm should also be of interest in parallel databases.

In conclusion, we observe that the GMDJ algorithm is a stable, flexible, and efficient algorithm for evaluating subqueries. Additional research is required on determining an optimal handling of non-neighboring predicates, something that has been ignored in the existing literature, and in the development of a database query engine that integrates GMDJs for handling subqueries.

References

[1] M. O. Akinde. Complex OLAP and distributed data warehousing. Ph.D. Thesis. To be published, 2003.

[2] M. O. Akinde and M. H. Böhlen. Generalized MD-joins: Evaluation and reduction to SQL. In *DB in Telecomm. II (VLDB-01)*, volume 2209 of *LNCS*, pages 52–67, Rome, Italy, Sept. 2001.

[3] M. O. Akinde, M. H. Böhlen, T. Johnson, L. V. S. Lakshmanan, and D. Srivastava. Efficient OLAP query processing in distributed data warehouses. In *EDBT'2002*, pages 336–353, Prague, Czech Republic, Mar. 2002.

[4] M. O. Akinde and O. G. Jensen. Maintenance and computation of complex aggregate views. Master's thesis, Aalborg University, Denmark, June 1998.

[5] L. Bækgaard and L. Mark. Incremental computation of nested relational query expressions. *ACM TODS*, 20(2):111–148, June 1995.

[6] G. v. Bültingsloewen. Translating and optimizing SQL queries having aggregates. In *VLDB'87*, pages 235–243, Brighton, England, UK, Sept. 1987.

[7] P. Celis and H. Zeller. Subquery elimination: A complete unnesting algorithm for an extended relational algebra. In *ICDE'97*, page 321, Birmingham, U.K., Apr. 1997.

[8] S. Ceri and G. Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE TSE*, 11(4):324–345, Apr. 1985.

[9] D. Chatziantoniou. Evaluation of ad hoc olap: In-place computation. In *SSDBM'99*, pages 34–43, Cleveland, Ohio, USA, July 1999.

[10] D. Chatziantoniou, M. O. Akinde, T. Johnson, and S. Kim. MD-join: an operator for complex OLAP. In *ICDE'2001*, pages 524–533, Heidelberg, Germany, Apr. 2001.

[11] D. Chatziantoniou and K. A. Ross. Groupwise processing of relational queries. In *VLDB'97*, pages 476–485, Athens, Greece, Aug. 1997.

[12] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *VLDB'87*, pages 197–208, Brighton, England, Sept. 1987.

[13] L. Fegaras. Query unnesting in object-oriented databases. In *SIGMOD'98*, pages 49–60, Seattle, Washington, USA, June 1998.

[14] C. A. Galindo-Legaria and M. M. Joshi. Orthogonal optimization of subqueries and aggregation. In *VLDB'2001*, Santa Barbara, California, USA, May 2001.

[15] R. A. Ganski and H. K. T. Wong. Optimization of nested SQL queries revisited. In *SIGMOD'87*, pages 23–33, San Francisco, California, USA, May 1987.

[16] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE'96*, pages 152–159, New Orleans, Louisiana, USA, Feb. 1996.

[17] W. Kim. On optimizing an SQL-like nested query. *ACM TODS*, 7(3):443–469, Sept. 1982.

[18] J. Lin and Z. M. Özsoyoglu. Processing OODB queries by O-algebra. In *CIKM'96*, pages 134–142, Rockville, Maryland, USA, Nov. 1996.

[19] M. Muralikrishna. Optimization and dataflow algorithms for nested tree queries. In *VLDB'89*, pages 77–85, Amsterdam, Holland, Aug. 1989.

[20] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *VLDB'92*, pages 91–102, Vancouver, British Columbia, Canada, Aug. 1992.

[21] R. Nakano. Translation with optimization from relational calculus to relational algebra having aggregate functions. *ACM TODS*, 15(4):518–557, Dec. 1990.

[22] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *SIGMOD'92*, pages 39–48, San Diego, California, USA, June 1992.

[23] J. Rao and K. A. Ross. Reusing invariants: A new strategy for correlated queries. In *SIGMOD'98*, pages 37–48, Seattle, Washington, USA, June 1998.

[24] P. Seshadri, H. Pirahesh, and T. Y. C. Leung. Complex query decorrelation. In *ICDE'96*, pages 450–458, New Orleans, Louisiana, USA, Feb. 1996.

[25] SQL Standards Comittee - American National Standards Institute. *ISO/IEC 9075:1999 Information technology - Database languages - SQL*, 1999.

[26] Q. Wang, D. Maier, and L. Shapiro. Algebraic unnesting of nested object queries. Technical Report CSE-99-013, Oregon Graduate Institute, 1999.