Repairing OLAP Queries in Databases with Referential Integrity Errors *

Javier García-García
IPN** / UNAM***
Mexico City, Mexico
Carlos Ordonez
University of Houston
Houston, TX, USA

***Abstract*—Many database applications and OLAP tools dynamically generate SQL queries involving join operators and aggregate functions and send these queries to a database server for execution. This dynamically generated SQL code normally assumes the underlying tables and columns are clean and lacks the necessary robustness to deal with foreign keys with null and invalid or undefined values that are ubiquitous in databases with inconsistent or incomplete content. The outcome is that at query time, several issues arise mostly as inconsistencies in answer sets, difficult to detect and explain by users of OLAP tools. In this article, we present an automated query rewriting method for automatically generated OLAP queries that are executed over tables with foreign key columns having potentially null or invalid values. Our method is applicable in queries that use join operators and aggregate functions obeying the summarizability property (e.g. $sum()$, $count()$). If a user of an OLAP tool wants or requests it, using our method the queries that use join operators may be rewritten and he or she may be warned of the referential integrity condition of the underlying database and the answer sets may present alternative consistent results in the case aggregate functions are involved. Preliminary experimental evaluation shows rewritten queries provide valuable information on referential integrity and take almost the same time as original queries, highlighting efficiency is good and overhead is minimal.**

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Systems - *Query processing*

General Terms

Algorithms, Experimentation, Performance

Keywords

DBMS, SQL

I. Introduction

Databases with referential integrity errors commonly arise in scenarios where several organizations have their databases integrated, where exchanging or updating information is frequent, or where table definitions change. Source databases may violate referential integrity and their integration may uncover additional referential integrity problems. In a data warehousing environment it is essential to repair referential integrity errors as early as possible in the ETL process, as it is recommended in [8]. A common strategy to repair referential integrity errors is to substitute invalid references with a "valid one" that refers to a row in the dimension table that is marked explicitly as undefined or not available (e.g. NA or 99). This solution, although it repairs the referential error, does not help much if the user wants to compute particular aggregated groups of answer sets of aggregate functions because valuable information that really belongs to "real" groups is lost in an undefined group [5]. Database applications and OLAP tools have to deal with these realistic databases. Applications may dynamically generate SQL queries and send them for execution to databases using interfaces such as JDBC. Developers usually pay scant attention to foreign keys with null and invalid or undefined values that are ubiquitous in databases with inconsistent or incomplete content. The outcome is that at query time, several issues arise mostly as inconsistencies in answer sets, difficult to detect and explain by users of OLAP tools.

In this work we propose a query rewriting method to be included in OLAP tools that may allow a user to request a referential integrity evaluation or an estimated answer set in the case the SQL code calls aggregate functions. The method consists in rewriting queries that use joins where foreign keys and equalities are involved. Queries with equijoins using foreign keys are the most common OLAP queries. We materialize a pre-aggregate temporal table which is used to obtain the requested answer set and appropriate data related to referential integrity problems. The user may request an evaluation and the system can send statistics about the referential integrity condition of underlying tables. An OLAP tool, when precomputing a cube, can also use our method to detect referential integrity problems.

The article is organized as follows. Section II contains definitions for join computation. Section III explains the generated SQL queries and how these queries are rewritten in order to consider referential integrity issues. Section IV contains an experimental evaluation over a synthetic database. Related work is discussed and compared in Section V. The article concludes with Section VI.

*This is the authors version of the work. The official version of this article was published in Proc. ACM Workshop on Data Warehousing and OLAP (DOLAP, CIKM 2010 Workshop), p.61-66, 2010.

**Instituto Politécnico Nacional
***Universidad Nacional Autónoma de México

II. Definitions

To provide a formal framework of study we use relational algebra notation for join operations, but queries are evaluated with SQL statements in the experimental section.

A. Referential Integrity

A relational database is denoted by $D(\mathcal{R}, I)$, where $\mathcal{R}$ is a set of N tables $\mathcal{R} = \{T_1, T_2, \ldots, T_N\}$, T_i is a set of rows and I a set of referential integrity constraints. A referential integrity constraint, belonging to I, between two tables T_i and T_j is a statement of the form: $T_i(K) \rightarrow T_j(K)$, where T_i is the referencing table, T_j is the referenced table, K is a *foreign key* (FK) in T_i and K is the primary key or a candidate key of T_j. In general, we refer to K as the primary key of T_j. To simplify exposition, we assume simple primary and foreign keys and the common attribute K has the same name on both tables T_i and T_j.

Let $t_i \in T_i$, then $t_i[K]$ is a restriction of t_i to K. In a valid database state with respect to I, the following two conditions hold for every referential constraint: (1) $T_i.K$ and $T_j.K$ have the same domains. (2) for every row $t_i \in T_i$ there must exist a row $t_j \in T_j$ such that $t_i[K] = t_j[K]$. The primary key of a Table ($T_j.K$ in this case) is not allowed to have nulls. But in general, for practical reasons the foreign key $T_i.K$ is allowed to have nulls when its value is not available at the time of insertion or when rows from the referenced table are deleted and foreign keys are nullified [3]. We refer to the valid state just defined as a *strict state*. In a data warehouse environment, a commonly used strategy to avoid joins that return answer sets with excluded rows is by inserting records into dimension tables to explicitly indicate not available or undefined references. When two tables are joined and aggregations are computed, rows with an undefined foreign key value that reference the undefined dimension row are aggregated in a group marked as undefined. Since this strategy effectively discards potentially valuable information, we also define a *rigorous state*. A rigorous state is a strict state where all dimension rows are defined, as opposed to the undefined dimension rows explained above.

B. Join on a Foreign Key

In this article, we concentrate on computing queries with natural joins between two tables linked by a foreign key and possibly with aggregate functions. We focus on joins with a simple key, consisting of one column. More specifically, we study join computation between two tables T_1, T_2 on a common column K: $T_1 \bowtie_K T_2$. Table T_1 may or may not have a measure attribute, say M, an attribute over which aggregate functions may be computed. The primary key of a table is underlined (e.g. $T_1(\underline{A}, K, M)$). This relational operation is called a natural join and it is the most common type of join in a relational database. For shorthand, we call this operation a FK/PK join. In an OLAP (multidimensional) database model in the form of a star schema for a data warehouse, there are two kinds of tables: a fact table and multiple surrounding dimension tables referenced by foreign keys $K_1, \ldots, K_k$. We focus on star-joins to compute aggregations. That is, computing joins between the fact table and several dimension tables.

$T_1 =$

A	K	M
1	7	10
2	3	12
3	2	23
4	1	13
5	null	22
6	2	34
7	1	75
8	null	null
9	99	7

$T_2 =$

K	B
1	One
2	Two
3	Three
99	Not Avail.

$G_1 =$

B	$sum(M)$
One	88
Two	57
Three	12
Not Avail.	7

$F =$

A	K	B
2	3	Three
3	2	Two
4	1	One
6	2	Two
7	1	One
9	99	Not Avail.

$G_2 =$

$sum(M)$
157

Fig. 1: Example of T_1, T_2, F, G_1 and G_2 .

C. Table Definitions and Aggregations

We now provide table definitions for T_1 and T_2 to compute queries with natural joins and aggregate functions. For a FK/PK join T_1 is defined as $T_1(\underline{A}, K, M)$ with K being a foreign key, possibly with duplicate, null, invalid or undefined values, and T_2 as $T_2(\underline{K}, B)$, where primary key $T_2.K$ has at every moment all valid values foreign key $T_1.K$ may hold. In an OLAP environment, T_1 could be the fact table and T_2 a dimension table. Let Table F store the join query results: $F = T_1 \bowtie_K T_2$. Result Table F shares the same primary key with T_1 and therefore it is defined as $F(\underline{A}, K, B)$. Let Table G_1 store the join group by query results with the aggregate answer set in case an aggregate function is computed over the foreign key K or an attribute determined by K, say B. Let Table G_2 store the total aggregate answer set. Figure 1 shows an example illustrating definitions. Table T_1 contains one row with an invalid key, two rows with null keys, and one with an undefined key. Result Table F does not contain any row with null or invalid foreign keys. In Table G_1 we are assuming the aggregate function $sum()$ was computed over the joined Table $T_1 \bowtie_K T_2$ grouping over attribute B and aggregating attribute M. Finally, in Table G_2 we find the total aggregate considering rows with a valid, defined foreign key. Rows in Figure 1 were rearranged for a better layout.

In our work, we assume aggregate functions are summarizable [9], [7]. A summarizable aggregate function is a distributive aggregate function that applied to an attribute is equal to a function applied to aggregates, that, in turn, are generated by the original aggregate function applied over the attribute of each partition of a table. Common aggregate functions such as count(*), count(), sum(), max(), min() meet this property. Other algebraic aggregate functions can be computed from these summarizable aggregates like, for instance, the avg() function, so our method may be used also with these kind of aggregates.

III. Repairing SQL Queries with

REFERENTIAL INTEGRITY ISSUES

OLAP tools generate SQL code to obtain answer sets like for instance F, G_1 or G_2 of our running example. Suppose the application was developed in Java. The Java code that could have generated the SQL query to obtain tables F and G_1 could be the following:

```
ResultSet getLabels(String refing, String refed) {
String joinquery = “SELECT ”
+ “A, K, B FROM ” + refing + “ JOIN ” + refed
+ “ ON ” + refing + “.K = ” + refed + “.K ;” ;
return sqlclause.executeQ(joinquery);
}
```

```
ResultSet getJoinAgg(String refing, String refed,
String agg) {
String joinaggquery = “SELECT ”
+ “ B, ” + agg +“(M) FROM ” + refing + “ JOIN ”
+ refed + “ ON ” + refing + “.K = ” + refed + “.K ”
+ “ GROUP BY B ;”;
return sqlclause.executeQ(joinaggquery);
}
```

Observe that the execution of the SQL code depends on the state of the database the moment the Java object code is executed. At the Java compile time, nothing is known about the database. At execution time, the user of the Java application may be ignorant about the database state, particularly about referential integrity issues. Rows 1, 5 and 8 of Table T_1 are effectively ignored. Data of row 9 do not participate in any “real” group, see Figure 1. Observe there are rows with referential errors.

A. Example

We illustrate the rewriting approach with the next example. Suppose the database application generates the following query:

```
SELECT B, sum(M)
  FROM T1 JOIN T2 ON T1.K = T2.K
  GROUP BY B;
```

Here the user wants a grouping by attribute B, the dimension description. This query corresponds to the one that produced Table G_1 of our running example. As we have seen, three rows were discarded because of an invalid value: value 7 and two null values in foreign key $T_1.K$. Also an undefined group was created. The user may consider unacceptable the fact that several rows were ignored or the existence of an undefined group holding valuable data.

The following SQL statements obtain the same answer set

```
CREATE TABLE temp1 AS
SELECT T2.B, T2.K AS FK,
  count(*) AS rowss,
  count(T1.K) AS diffnull,
  count(T2.K) AS validv,
  sum(M) AS agg1
FROM T1 LEFT OUTER JOIN T2
  ON T1.K = T2.K
GROUP BY B, FK;

SELECT B, sum(agg1)
FROM
(SELECT B, agg1
FROM temp1
WHERE temp1.FK is not null) AS foo
GROUP BY B;
```

TABLE I: Contents of Table $temp1$

B	FK	$rowss$	$diffnull$	$validv$	$agg1$
One	1	2	2	2	88
Two	2	2	2	2	57
Three	3	1	1	1	12
Not Avail.	99	1	1	1	7
null	null	3	1	0	32

The following small variant of the last SQL statement will obtain Table G_2 from Table $temp1$

```
SELECT sum(agg1)
FROM
(SELECT B, agg1
FROM temp1
WHERE temp1.FK is not null
  AND temp1.FK <> 99) AS foo;
```

However, we have a table, $temp1$, that holds valuable data that can be used to determine the condition of the foreign key K, with respect to referential integrity. Also, it holds information related to the undefined value (99). In Table I we show the contents of Table $temp1$. This table is normally much smaller than the referencing table since it holds the different valid values of foreign key $T_1.K$ including the undefined value, plus a row with a null value in FK that represents rows with an invalid value, assuming null is invalid. From this row we can obtain the number of invalid values different from null. Notice that the overhead to compute the answer set by creating Table $temp1$ consists only in a double scan over Table $temp1$. The double scan is because values of attribute C may be duplicated. If this is not the case, only one scan is needed.

B. Query Rewriting Method

To generalize the previous example we proceed as follows. Given a star-join query of the form:

```
SELECT attr1, attr2,..., attrn,
       agg1(M1), agg2(M2),...,aggm(Mm)
  FROM T1 JOIN T2 ON T1.K2 = T2.K2
          JOIN T3 ON T1.K3 = T3.K3
          JOIN,...,
           Tk ON T1.Kk = Tk.Kk
  GROUP BY attr1, attr2,..., attrn;
```

where T_1 is the referencing table or a fact table and $T_2, \ldots, T_k$ are the referenced tables or dimension tables. $K_2, \ldots, K_k$ are foreign keys that correspond to the primary keys in tables $T_2, \ldots, T_k$ respectively. $attr_1, attr_2, \ldots,$ $attr_n$ are attributes in tables in set $\{T_1, \ldots, T_k\}$. $agg1(M_1)$, $agg2(M_2)$,..., $aggm(M_m)$ are aggregate functions taken from {count(*), count(), sum(), avg(), max() and min()} over (measure) attributes of Table T_1.

The rewriting is the following

```
CREATE TABLE temp1 AS
SELECT attr1 AS C1, attr2 AS C2,...,
        attrn AS Cn,
        T2.K2 AS FK2,
        T3.K3 AS FK3,...,Tk.Kk AS FKk
  count(*) AS rowss,
  count(T1.K2) AS diffnull2,
```

```
    count(T2.K2) AS validv2,
    count(T1.K3) AS diffnull3,
    count(T3.K3) AS validv3,...,
    count(T1.Kk) AS diffnullk,
    count(Tk.Kk) AS validvk,
    agg1(M1) AS agg1, agg2(M2) AS agg2,...,
    aggm(Mm) AS aggm
FROM
T1 LEFT OUTER JOIN T2 ON T1.K2 = T2.K2
   LEFT OUTER JOIN T3 ON T1.K3 = T3.K3...
   LEFT OUTER JOIN Tk ON T1.Kk = Tk.Kk
GROUP BY C1,C2,...,Cn, FK2,FK3,...,FKk;

SELECT C1 AS attr1, C2 AS attr2,...,
    Cn AS attrn,
    func1(agg1), func2(agg2),...,
        funcm(aggm)
FROM
(SELECT C1, C2,...,Cn, agg1, agg2,...,
        aggm
FROM temp1
WHERE temp1.FK2 is not null AND
        temp1.FK3 is not null AND...AND
        temp1.FKk is not null) AS foo
GROUP BY C1,C2,...,Cn;
```

The rewritten variant to obtain the total aggregate, like Table G_2 in our running example is similar. Notice that the summarizable property is important since it allows us to compute the answer set in two steps. First we compute Table $temp1$ with a precomputation of the aggregate functions and the information related to the referential integrity conditions. Next, we finish the computation of the answer set. In parallel, with data in Table $temp1$, if there is a row with a null value in column FK we know there are referential integrity issues. Using columns $rowss$, $diffnull_i$ and $validv_i$ the system can compute the number of rows of a given group including the null group (rows with invalid values in foreign keys) and the undefined group, and for each foreign key, the number of rows with a value different from null as well as rows with valid values. This way a query with referential integrity issues is detected. Observe that the aggregate computation is *safe* in the sense that if there are no referential integrity errors, that is, if the database is a strict database, the row with a null value in FK in Table $temp1$ will not appear, and the answer set will be correct. Notice that OLAP processing can be slow, but techniques [10], such as precomputation by aggregating on all dimensions, help improve performance. Our method can be used in the precomputation of cubes or in the ETL process in order to detect referential integrity issues. It can be applied on cube exploration (slice/dice, pivoting, x tab).

Aggregate functions, aside from the sum() aggregate shown in our example, such as count(*), count(), max(), min() and avg() can also be computed with this method. Aggregate function count(*) derives from column $rowss$, adding (sum()) partial counts. Aggregate count() can be computed from column $validv$. To compute max() and min() just we need to add these aggregates in Table $temp1$. Function avg() can be computed with functions sum() and count(). If the user requests it, undefined rows may be omitted and the aggregate may be computed using a referential partial probability vector as it is explained in [5] in order to dynamically estimate the answer set of the aggregation functions. This way, the user will get an estimated answer set of a rigorous database.

A variant of the computation of Table $temp1$ presented above, consists in computing first a table grouping the referencing (fact) table, by the set of foreign keys. That is, on all dimensions in order to help improve performance when computing equijoins. This reduced table that may or may not be materialized, can hold aggregate computations that correspond to the number of rows, the number of rows with foreign keys different from null and the measure attributes. This table holds extra rows and aggregations that correspond to invalid foreign key values. We proceed then to compute with this table left outer joins with the corresponding dimension tables in order to obtain the number of rows with valid values and values that correspond to attributes of dimension tables (attribute B in Table I).

TABLE II: PDFs used to insert invalid values.

PDF	$Probability\ function$	$Parameters$
Uniform	$\frac{1}{h}$	$h = \|T_2\|$
Zipf	$\frac{1/k^s}{H_{M,s}}$	$M = \|T_2\|$, $s = 1$
Geometric	$(1-p)^{n-1}p$	$p = 1/2$

Finally, observe that if there is a small fraction of referential errors, a left outer join should use the same join algorithm as an equijoin (mergesort join, hash join or indexed join) therefore our method is efficient in these cases.

IV. Experimental Evaluation

We conducted our experiments on a database server with one CPU running at 1.6 GHz with 2 GB of main memory and 146 GB on disk. Evaluations were carried out on the public domain DBMS PostgreSQL.

A. TPC-H Database

Our synthetic databases were generated by the TPC-H DBGEN program. We did not define any referential integrity constraint to allow referential errors. We inserted referential integrity errors in the referencing fact table (lineitem) with different rates of errors and in three foreign keys ($l\_orderkey$, $l\_partkey$ and $l\_suppkey$).

The referencing table, $lineitem$ has cardinalities 6M and 12M, referenced tables $orders$, $part$ and $supplier$ have the following cardinalities: 1.5M, 200k and 10k rows, respectively. Invalid values were randomly inserted according to three different pdfs, that follow the parameters shown in Table II where T_2 stands for the referenced table. The minimum number of errors generated was approximately 6,000 and the maximum 600,000. One value of each set of valid values was considered the $Not\ Available$ group.

B. Time Performance

We evaluated two optimization variants to compute the preaggregated table $temp1$ described in Section III-B. The first variant, left outer join first (LOJF), obtains $temp1$ by first computing left outer joins between the referencing (fact) table and the referenced (dimension) tables. Afterwards the grouping computation of compound foreign keys takes place.

TABLE III: Table $temp1$ computation with different compound foreign keys groupings. Time in seconds.

FK (group size)	$LOJF\ variant$		$GF\ variant$	
$\lvert lineitem \rvert$	$6M$	$12M$	$6M$	$12M$
$l\_suppkey$ (600)	134	27	113	343
$l\_partkey$ (30)	143	254	247	517
$l\_orderkey$ (4)	133	276	253	528
$l\_suppkey, l\_partkey$ (7,5 avg)	236	396	202	519

The SQL rewrite template of this variant is the one that appears in Section III-B. The second variant, grouping first (GF), computes $temp1$ by first grouping the compound foreign keys of the referencing table. This is done to obtain aggregate computations first, without considering the referential integrity errors. Then follows the computation of left outer joins. The second variant prove to be more efficient when the number of distinct values of the compound foreign keys were significantly less than the cardinality of the referencing table. Otherwise, computing left outer joins first was the best option.

In Table III we present performance considering compound foreign key value groups of different sizes (meaning each compound foreign key value appeared in a number of rows on average). Observe how times become better for groups of larger size (with more rows) when using the GF variant.

Summarizing the performance of our proposal, computation depends on the size of the referencing table, the size of groups of compound foreign key values and the number of distinct compound foreign key values.

V. Related Work

There are several articles that study the quality of dynamically created SQL queries [2], [6]. In [6] the authors present a technique that considers data type errors, however referential integrity is not taken into consideration. In [2] the purpose is to analyze relationships between tables, among other structures, and to extract programmatic joins and cascading deletes, and summarize data access. Also, several metrics are defined for quantifying quality aspects of systems that contain embedded SQL queries. In [1] the authors present a consistency check model that can handle a subset of SQL. The model handles foreign keys via the NOT EXISTS SQL set operator. Although the user receives a warning, the system does not give a clue about the size of the problem. Moreover, aggregate functions are not considered. Several other articles study SQL query consistency taking into consideration referential integrity, such as [12]. In contrast, our proposal considers embedded SQL and aggregate functions.

The process of finding a query rewrite of a cube view as another query that uses a precomputed cube view is known as aggregate navigation [8]. Summarizability has been studied in the OLAP context, particularly in the aggregate navigation process, to compute cube views more efficiently [9]. However, to the best of our knowledge, the use of summarizability in the aggregate navigation process has not been used to obtain data quality information, specifically to detect referential integrity issues.

Next, we summarize past research on improving database systems to handle referential integrity issues. In [11] we propose measuring referential integrity errors. In [4] we consider an additional metric to measure consistency in table replicas. In [5] we presented our studies of how to improve aggregations. In contrast, in this work we study how to rewrite queries with potential referential integrity violations, generated by OLAP tools. Here we do not compute a new extended aggregation as in [5]. By using the summarizable property of certain aggregations, during the query generation process, we collect valuable information and we use it to diagnose data quality issues and warn the user about correctness of referential integrity in a queried table.

VI. Conclusions and Future Work

We proposed a query rewriting method to repair dynamically generated SQL queries combining join operators and aggregate functions, in order to detect referential integrity issues hidden in the underlying database. Our method takes advantage of the summarizability property of common distributed aggregate functions such as $count(*)$, $count()$, $sum()$, $max()$ and $min()$, to generate statistics related to referential integrity issues. With referential integrity related data, the system is able to warn the user about referential integrity in queried tables. Observe that our proposed method can be easily incorporated into OLAP tools. The overhead to compute the additional referential integrity related statistics has a low impact in overall performance. Our query rewriting method can be incorporated in the precomputation of cube views. Our experiments show the proposed method scales well.

There are several issues for future work. Some of our ideas can be extended to other non-summarizable aggregates and other types of SPJ queries. We need to study query optimization with compound keys in more depth. Also, we need to study how to efficiently reuse referential integrity statistics in future queries. We are currently exploring how to apply our techniques over materialized views during the query computation.

Acknowledgments. The first author was partially supported by the IPN - Secretaría de Investigación y Posgrado. The second author was partially supported by NSF grants CCF 0937562 and IIS 0914861.

References

[1] Stefan Brass and Christian Goldberg. Proving the safety of SQL queries. In *QSIC '05*, pages 197–204, 2005.

[2] H. Brink, R. Leek, and J. Visser. Quality assessment for embedded SQL. In *SCAM '07*, pages 163–170, 2007.

[3] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison/Wesley, Redwood City, California, 3rd edition, 2000.

[4] J. García-García and C. Ordonez. Consistency-aware evaluation of OLAP queries in replicated data warehouses. In *ACM DOLAP '09*, pages 73–80, 2009.

[5] J. García-García and C. Ordonez. Extended aggregations for databases with referential integrity issues. *Data Knowl. Eng.*, 69(1):73–95, 2010.

[6] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *ICSE '04*, pages 645–654, 2004.

[7] J. Horner, I. Song, and P. Chen. An analysis of additivity in OLAP systems. In *DOLAP '04*, pages 83–91, 2004.

[8] R. Kimball and J. Caserta. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data*. John Wiley & Sons, 2004.

[9] H. J. Lenz and A. Shoshani. Summarizability in OLAP and statistical data bases. In *SSDBM Conference*, pages 132–143, 1997.

[10] C. Ordonez and Z. Chen. Evaluating statistical tests on OLAP cubes to compare degree of disease. *IEEE Trans. Info. Tech. Biomed.*, 13(5):756–765, 2009.

[11] C. Ordonez and J. García-García. Referential integrity quality metrics. *Decision Support Systems Journal*, 44(2):495–508, 2008.

[12] J. Tuya, M.J. Suárez-Cabal, and C. Riva. A practical guide to SQL white-box testing. *SIGPLAN Not.*, 41(4):36–41, 2006.