Evaluating Join Performance on Relational Database Systems

Carlos Ordonez
Department of Computer Science, University of Houston, Houston, USA
ordonez@cs.uh.edu

Javier García-García
UNAM/IPN, Mexico, Mexico
javgar@servidor.unam.mx

Received 25 October 2010; Revised 6 December 2010; Accepted 14 December 2010

The join operator is fundamental in relational database systems. Evaluating join queries on large tables is challenging because records need to be efficiently matched based on a given key. In this work, we analyze join queries in SQL with large tables in which a foreign key may be null, invalid or valid, given a referential integrity constraint. We conduct an extensive join performance evaluation on three DBMSs. Specifically, we study join queries varying table sizes, row size and key probabilistic distribution, inserting null, invalid or valid foreign key values. We also benchmark three well-known query optimizations: view materialization, secondary index and join reordering. Our experiments show certain optimizations perform well across DBMSs, whereas other optimizations depend on the DBMS architecture.

General Terms: query optimization; performance evaluation

Additional Key Words and Phrases: Join Processing, Foreign Key, RDBMS

1. INTRODUCTION

The join operator is fundamental in query processing in a relational database management system (RDBMS). In this work we benchmark the join operator on modern relational database systems, using state of the art join algorithms. In an OLTP database where referential integrity is enforced, foreign keys have valid values. Therefore, most join queries work with valid keys. However, for practical reasons, to allow record insertion when there is missing information or when the logical data model behind the database evolves, foreign keys may be allowed to have null values; we will discuss this case in detail later. On the other hand, when multiple databases

Copyright(c)2010 by The Korean Institute of Information Scientists and Engineers (KIISE). Permission to make digital or hard copies of part or all 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. Permission to post author-prepared versions of the work on author's personal web pages or on the noncommercial servers of their employer is granted without fee provided that the KIISE citation and notice of the copyright are included. Copyrights for components of this work owned by authors other than KIISE must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires an explicit prior permission and/or a fee. Request permission to republish from: JCSE Editorial Office, KIISE. FAX +82 2 521 1352 or email office@kiise.org. The Office must receive a signed hard copy of the Copyright form.

are integrated into a central database, or when transactional databases, from different organizations, exchange information foreign keys will likely have invalid values. An end user may be confronted with the problem of computing join queries with tables having inconsistent keys. This problem may be compounded by denormalization and replication, when some, but not all, tables are replicated in a distributed environment. Denormalization propagates referential issues. Replication is generally used to keep local copies of tables to avoid remote network access, but it introduces potential inconsistency. This article studies join queries with null, invalid and valid keys, considering referential integrity. That is, we consider all potential cases: a key that should be ignored (null), a key that will result in an unsuccessful search (invalid) and a key that results in a successful search and will produce a match and an output row (valid). Foreign keys involving nulls and invalid values are typical in database integration, ETL (Extract, Transform, Load) scripts in a data warehouse and data mining preparation. On the other hand, foreign keys having only valid values are representative of OLTP databases where referential integrity is enforced. Notice join queries on tables with null and invalid keys are considered related, but algorithmically different, because null keys do not participate in join processing, whereas invalid keys do result in an unsuccessful search (and failed key match). We identify factors that impact join performance and we analyze well-known SQL query optimizations. Studied factors include table size, row size, key column cardinality, and key probability distribution. On the query optimization side we consider view materialization, indexing foreign keys and join reordering. Based on our findings, we give several recommendations.

The article is organized as follows. Section 2 contains definitions. Section 3 explains synthetic data set generation and three well-known query optimizations. Section 4 contains an extensive experimental evaluation on three RDBMSs. Related work is discussed in Section 5. The article concludes with Section 6.

2. DEFINITIONS

To provide a formal exposition we use relational algebra notation to explain join queries.

2.1 Join on a Foreign Key

We focus on computing natural joins between two tables linked by a key, which is a common query in practice. More specifically, we study natural join computation between two tables T_1, T_2 on a common column K: $T_1 \bowtie_K T_2$, such that K is a foreign key in T_1 and a primary key in T_2.

We call T_1 the referencing table and T_2 the referenced table. Column $T_1.K$ is allowed to have invalid "foreign key" values. That is, there exist values in $T_1.K$ which are not in $T_2.K$. Also, K may include nulls. Let table R store the natural join query results: $R = T_1 \bowtie_K T_2$. The size of table T_i is denoted by $|T_i|$. The primary key of a table is underlined (e.g., $T_1(\underline{A}, K)$).

2.2 Table Definitions

We define tables T_1 and T_2 to compute natural joins. Let A, K, X_i, Y_j ($i = 1 \ldots p$, $j =$

$T_1 =$

A	K	X_1	X_2
1	7	0	1
2	3	1	1
3	2	1	0
4	1	1	1
5	null	1	1
6	2	0	0
7	1	0	1
8	null	0	0

$T_2 =$

K	Y_1
1	1
2	2
3	3
4	4

$R =$

A	K	X_1	X_2	Y_1
2	3	1	1	3
3	2	1	0	2
4	1	1	1	1
6	2	0	0	2
7	1	0	1	1

Figure 1. Example of T_1, T_2 and R.

1 ... q) be columns of integer type (we prefer integers for simplicity of implementation and speed). T_1 is defined as $T_1(\underline{A}, K, X_1, ..., X_p)$ with K being a foreign key, possibly with duplicate values, and T_2 as $T_2(\underline{K}, Y_1, ..., Y_q)$, where K is a primary key. Columns X_1, ..., X_p, Y_1, ..., Y_q act as non-key columns and are used to simulate large records in experiments.

2.3 Example

Figure 1 shows a small example illustrating the definitions above. There are a few referential integrity issues in T_1. Table T_1 contains one row with an invalid key and two rows with null keys. The join result table R does not contain any row with invalid keys. Table T_2 has rows not referenced by T_1, but that is acceptable from a referential integrity point of view.

3. BENCHMARKING JOIN QUERIES

This section explains our experimental setup in abstract terms. We contrast difierent kinds of join operations based on a foreign key. We explain how benchmarking tables are generated. We then explain well-known query optimizations.

3.1 Primary and Foreign Keys

Null and invalid key values create challenges for query optimization. Primary keys cannot be null or partially null [Elmasri and Navathe 2000; Codd 1979]. On the other hand, there exist three mutually exclusive cases for a foreign key value. We indicate the term used in the experimental section in parentheses. (1) a foreign key value exists (valid). (2) a foreign key value does not exist (invalid), violating referential integrity. (3) a foreign key value is marked as null (null), which is a compromise between the first two cases. Case 1 represents the ideal case. If referential integrity is enforced then all foreign key values are valid. Case 2 may happen on databases with relaxed referential integrity constraints or in integrated databases (e.g., a data warehouse). Case 3 is a common practical solution for missing information when a new record is inserted. Nulls can stand for: inapplicable or not available attribute values [Codd 1979]. In this work we assume they represent not available data since that is the most common scenario. That is, they represent missing information. We should stress that, in general, not applicable values are due to bad database model designs as explained in [Elmasri and Navathe 2000].

To provide a complementary view, we discuss nulls in the context of keys. Nulls obey common guidelines: (1) Nulls are sometimes allowed in a foreign key to allow updating tables with incomplete data. (2) Nulls are not allowed in a primary key K since a primary key is the fundamental mechanism to identify and search for a row. (3) Outer joins may produce nulls in foreign keys in tables where referential integrity is not enforced. That is, the optimizer may populate columns with nulls when tables to be joined have incomplete or inconsistent content.

3.2 Synthetic Data Generation

Data generation was performed entirely inside the RDBMS, in contrast to external synthetic database generators (e.g., DBGEN from TPC-H), which require generating external files and then importing them into the RDBMS. Relational queries in SQL provided a simple, fast and portable way to generate synthetic data and it gave us fiexibility to modify foreign key probabilistic distributions. However, it is also feasible to generate synthetic tables with a program in a traditional programming language like C++ and then import the files into the RDBMS. Our goal is to study how null, invalid and invalid keys impact join processing. Specifically, we want to understand which factor(s) have more weight on query evaluation time, including table sizes, physical row size, key cardinality, key distribution and join algorithms.

Our basic setup consists of having T_1 with an initial large number of rows all having valid foreign keys in K and then adding a Δ increment of rows with foreign key K values of the same kind (explained below) to T_1, keeping T_2 fixed. Each Δ row has either null, invalid or valid values for K, being an invalid value the default. Then we iteratively compute $R = T_1 \bowtie_K T_2$ to understand time trends (time complexity, I/O cost, scalability, overhead). Under this data generation scheme, R will remain constant for join queries when Δ contains null or invalid foreign key values since those additional rows are filtered out by the $\bowtie$ operator. Assuming Δ is $\cup$-compatible with T_1 and it contains only null or invalid values in K with respect to T_2 then $R = T_1 \bowtie_K T_2 = (T_1 \cup \Delta) \bowtie_K T_2$.

Generating row identifiers in a portable manner was challenging because each RDBMS provided particular functions to generate record identifiers. Also, even though OLAP window functions could be used to generate row identifiers they are slow and not supported by the public domain RDBMS. Instead we created row identifiers by a series of Cartesian products with a small table, simulating loops (*i.e.*, to create a 1 M row table we cross-joined a table with 10 rows 6 times). Such approach made data generation queries fast and portable.

For invalid keys we analyzed four fundamental discrete probabilistic distribution functions (pdfs) including the constant, the uniform, the zipf and the geometric distribution. These functions cover a wide spectrum of real-life scenarios, going from the "default" uniform case to the completely skewed case. The constant pdf has one invalid value repeated for the entire increment Δ; this pdf represents a completely skewed distribution. The uniform pdf draws values uniformly sampled from $|T_2|$ distinct *invalid* values; not all values may be represented if the sample is small. The probability of one value is given by $1/|T_2|$. The uniform pdf represents the classical case in query optimization and is well understood. Therefore, it is our default case.

The zipf distribution has invalid values having a linear sequence of frequencies: 1; 2; 3 and so on. That is, invalid values have an approximate probability $k/|\Delta|$ of appearing, for $k = 1, 2, ...$ Finally, the geometric distribution consists of invalid values having a sequence of geometrically growing frequencies: 1, 2, 4, and so on, given by 2^{k-1} for $k = 1, 2,....$ The zipf and geometric pdfs represent skewed distributions. We treat a null and one constant value differently, since nulls in K are automatically filtered out before join processing.

3.3 Query Optimizations

We evaluate three well-known query optimizations [Garcia-Molina et al. 2001]: (1) View materialization for join query ($T_1 \bowtie_K T_2$); (2) Secondary index on foreign key K in T_1. (3) Join reordering.

In the view materialization optimization we create a table S_1 which excludes invalid keys from T_1 and unreferenced keys from T_2: $S_1 = \pi_{T_1} (T_1 \bowtie_K T_2)$: The π projection operator is used to obtain the same definition for T_1 and S_1. This strategy has been successfully applied before in distributed query processing and it has been called semijoin. We should mention that most join evaluation techniques consider the case that many key values in T_2 will not be referenced, whereas we also consider that many T_1 values do not have valid references. View materialization represents an optimal strategy to evaluate join queries since it excludes any invalid keys, but it incurs on significant space usage. More importantly, the view needs to be incrementally maintained when new rows are inserted into T_1.

For the secondary index optimization an additional index on K only for T_1 is created. It is assumed T_1 already has an index on its primary key, but K may not be indexed. For integrated databases and databases exchanging information we expect K not to have an index in T_1. The secondary index uses much less space than a materialized view and it can be easily maintained, but it incurs on search overhead at query evaluation time.

The join reordering optimization is applicable when there are two or more joins involving foreign keys involving three or more tables. When there is a multi-way join, joins with a higher number of invalid values are evaluated first. This strategy attempts to reduce intermediate table sizes as early as possible. Join reordering is a well-known algebraic optimization which attempts to reduce the size of intermediate tables by evaluating joins that have higher number of invalid keys earlier, like traditional relational query optimization. This optimization relies on keeping extended statistics on foreign key columns. Such statistics can be easily maintained when the same join queries are evaluated over and over. In the experimental section we focus on joins between one table with several foreign keys and several referenced tables (i.e., similar to one fact table surrounded by dimension tables in an OLAP database).

4. EXPERIMENTAL EVALUATION

4.1 Setup

We used three modern RDBMSs (one open-source, two commercial), running on the following hardware configuration. We avoid mentioning the specific commercial name

of each RDBMS since our article does not intend to compare (i.e., benchmark) them with each other. Instead, we want to find general trends and evaluate query optimizations on state-of-the-art database systems. The three RDBMSs were running on three identical servers with a CPU at 3.2 GHz, 4 GB of main memory and 1 TB on disk. As explained in Section 3, synthetic table generation and performance evaluation was performed with automatically generated SQL code. We used relatively small tables (relative to RAM) in order to repeat queries many times, but we cleared the RDBMS cache memory before each experiment using RDBMS-specific commands (i.e., tables were always read from disk). Most experiments use the default index on the primary key provided by each RDBMS without secondary indexes, which are studied separately. Times are measured in seconds.

As explained in Section 3, R is appended with Δ increments of keys, where all keys are of the same kind. We decided not to insert sets of rows having a mix of nulls and invalid keys because nulls are semantically different from invalid key values and because they are internally manipulated by each RDBMS in a different manner. Invalid keys have a uniform distribution by default. But we also study the impact of other probability distribution functions. When all keys in Δ are null or invalid the join result R remains the same (i.e., it remains constant across queries). On the other hand, when foreign keys in Δ are valid then R is different on each query.

4.2 Performance Evaluation Issues

Time measurements had significant variability depending on the specific setup for query evaluation. There are two well-known facts in query evaluation. First, evaluating a query in temporary disk space is much faster than evaluating it in permanent disk space (saving results in a table). Second, consecutive repetitions of the same query accelerates evaluation due to buffer warming. We experimentally compared temporary and permanent disk space speedup and query acceleration due to repetition, but due to lack of space we do not show times in the paper. Evaluating the join in temporary disk space was significantly faster (two to three times faster than permanent space). Then we analyzed the effect of buffer warming by repeating the same query five times. It produced a significant acceleration in RDBMS A (1/3 time), good acceleration in RDBMS C (1/2 time) and no acceleration in RDBMS B (no table caching). Therefore, in order to make an accurate performance evaluation, we evaluated joins in temporary disk space and avoided buffer warming by dropping and re-creating T_1 and T_2 before benchmarking each join query.

Tables T_1 and T_2 were indexed as follows. We used the default indexing mechanism

Table I. Join algorithm.

second. idx	$\|T_2\|$	RDBMS A	RDBMS B	RDBMS C
N	1k	hash	hash	hash
N	100k	sort-merge	hash	hash
Y	1k	sort-merge	hash	sort-merge
Y	100k	sort-merge	hash	sort-merge

of each RDBMS specifying a primary key (PK) constraint in the DDL (table definition). Based on their PK, both tables T_1 and T_2 had a primary key index by default. The foreign key K in T_1 did not have a secondary index by default. The last set of experiments evaluates the impact of a secondary index.

In summary, our default setup was as follows. We measured elapsed time for execution of each query five times, and we report the average, indicating the elapsed time in seconds. In general time measurement variability was small for each set of experiments. Queries were evaluated in temporary space and tables were dropped and recreated before time measurement to avoid caching. In general, the tables were generated with parameters $|T_1| = 1$ M, $|\Delta| = 1$ M, $|T_2| = 100$k, $p = 10$, $q = 10$.

4.3 RDBMS Query Plans

We analyzed the query plans based on experiments parameters $|T_1|$, $|T_2|$, Δ, pdf and secondary index on K in T_1. We found that the number of rows with null or invalid keys did not influence the query plan. The two main parameters that influenced choice of a query plan were table size and secondary index. Hash-joins and sort-merge joins were the two join algorithms used in all RDBMSs. That is, no RDBMS used a nested-loop join even when T_2 was small. For RDBMS A the size of table T_2 influenced the preferred join algorithm. For RDBMS B the join algorithm selected by the optimizer was always a hash join. In RDBMS C the secondary index of T_1 determined the join algorithm. In RDBMS A no column statistics are computed on a table by default; the user needs to explicitly collect statistics. In RDBMS B and RDBMS C statistics are automatically collected, but the overhead was lower in RDBMS C. Table I shows join algorithms selected by the optimizer depending on T_2 and the secondary index for T_1.

4.4 Impact of Invalid Keys

We start by studying the effect of $|T_2|$ on time. Table II shows how time varies as $|T_2|$ varies keeping T_1 fixed. In this case K in T_1 has a uniform pdf. We insert rows into T_2 that are not referenced by T_1. For the three RDBMSs the time to join nulls is less than the time to join invalid key values, which in turn is less than the time to join valid keys. To understand why, we analyzed the query plans: before joining tables, null keys are filtered out and valid keys end up producing an output row in R. In RDBMS A and RDBMS C we can see $|T_2|$ plays a minor role in join performance since times grow very slowly when T_2 grows an order of magnitude. In

Table II. Varying T_2, $|T_1|$=2M and $|\Delta|$=1M (n=null, i=invalid, v=valid; times in secs).

	RDBMS A			RDBMS B			RDBMS C		
$\|T_2\|$	n	i	v	n	i	v	n	i	v
1K	14	12	27	3	3	4	1.5	1.5	1.5
10K	15	15	27	3	3	4	1.5	1.5	1.5
100K	19	28	37	5	6	8	1.6	1.6	1.7
1,000K	22	29	43	11	13	15	1.6	1.7	1.7

Table III. Varying $|\Delta|/|T_1|$ and $|T_2|$ (n=null, i=invalid, v=valid; times in secs).

			RDBMS A			RDBMS B			RDBMS C		
$\|\Delta\|$	$\|T_1\|$	$\|T_2\|$	n	i	v	n	i	v	n	i	v
0	1,000K	1k	13.4	13.2	14.2	2.4	2.4	2.4	0.5	0.5	0.5
800K	1,800K	1k	14.2	14.8	25.2	2.8	2.8	3.0	1.2	1.5	1.7
1,600K	2,600K	1k	14.5	14.9	30.9	2.9	3.4	4.5	1.3	1.7	1.9
0	1,000K	100k	14.1	14.4	14.6	3.6	4.0	4.6	0.5	0.5	0.5
800K	1,800K	100k	20.7	24.8	41.1	4.7	6.7	8.1	1.4	1.4	1.6
1,600K	2,600K	100k	21.4	31.5	48.2	4.8	7.0	10.4	1.4	1.7	2.1

RDBMS A time barely doubles when T_2 grows three orders of magnitude, whereas for RDBMS C time growth is almost flat. In contrast, in RDBMS B there is a more significant relative time growth, where time doubles when T_2 grows an order of magnitude. Therefore, $|T_2|$ is more important in RDBMS B. In the remaining experiments we test join queries under challenging conditions, with $|T_2| = 100$k as default.

We now turn our attention to $|T_1|$, where T_1 has a growing number of rows with null/invalid keys. We study scalability experiments varying $|\Delta|$ when T_2 is small or large (e.g., 1k versus 100k). Both valid and invalid keys were uniformly distributed. Results are summarized in Table III. Ideally, if the RDBMS "knew" T_1 rows have invalid/null keys times should remain almost constant (i.e., the time for $|\Delta|=0$). When T_2 is small we can see in all RDBMSs nulls/invalids take almost the same time. In RDBMS C the three foreign key times are practically the same, stressing its efficiency. When T_2 is large, times do grow when $|T_1|$ increases and there is a gap between nulls and invalids: invalid key values take more time. In general, time grows linearly for invalids and sublinearly for nulls. In a similar manner to the previous experiment, joining valid keys takes more time than joining invalid keys, which is explained by the extra time needed to write output rows. We conclude that $|T_1|$ is more important than $|T_2|$, especially for invalid keys.

Figure 2 analyzes time growth varying $|T_1|$ for an initial $|T_1|$=1M. We compare with $|T_2|$=100k (large) and $|T_2|$=1000 (small), inserting $|\Delta|$ rows with null foreign key values. In RDBMS A time grows slowly for invalids when $|T_2|$=100k, but remains almost constant otherwise. In RDBMS B time is almost constant for nulls and grows faster for invalids. Time grows linearly in RDBMS C when T_2 is large, but grows slowly when T_2 is slow. We can see hash-joins scale well in all RDBMSs, showing nulls have little impact. On the other hand, we can see the sort-merge join (RDBMS A) is more impacted by nulls/invalids. We conclude that nulls marginally impact performance (as expected) and invalid keys indeed have an impact on performance.

Our next experiment analyzes the impact of row (record) size in both T_1 and T_2, leaving one or the other fixed, as shown in Table IV. Recall from Section 2 that p is row size of T_1 and q is the row size of T_2. Notice T_1 does not have an index on K and therefore every row is scanned. We vary p and q to understand which one plays a

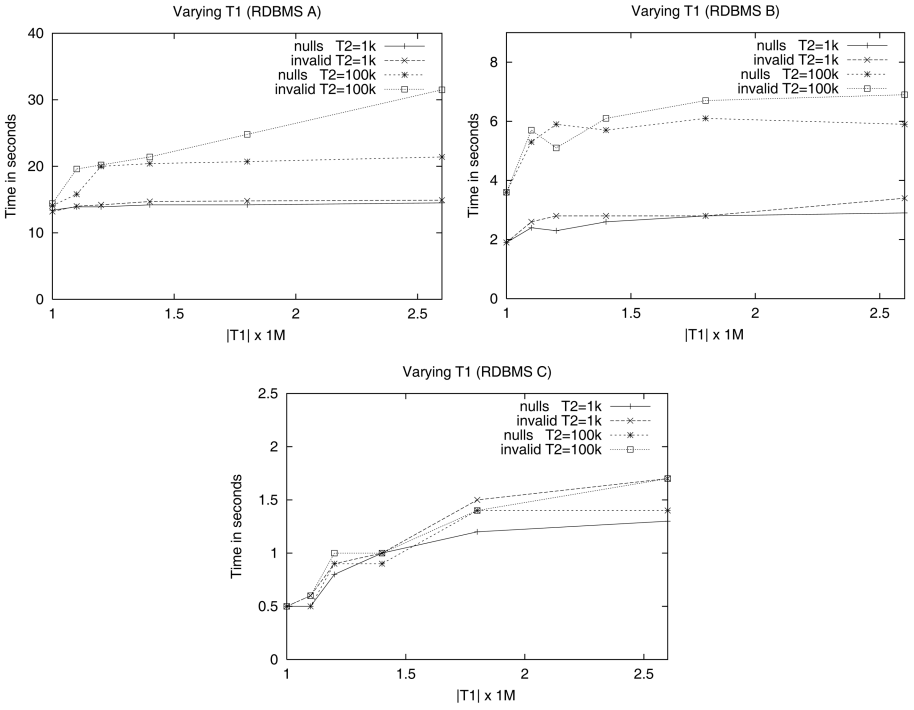

Fig. 2. Time growth varying $|T_1|$.

more important role in performance. In this case we iteratively duplicate row size to observe if I/O time doubles (i.e. row size grows geometrically). We can see times remain almost constant in RDBMS C, but they slowly grow in RDBMS A and RDBMS B, especially for invalids. However, they do not grow at the same geometric rate as p or q. The reason is I/O is block based and wider rows have a marginal impact on time. It is interesting times do grow when q grows, despite the fact that T_2 is indexed. We conclude that the impact of row size is RDBMS-specific and that there

Table IV. Varying p (row size of T_1) and q (row size of T_2), $|\Delta|$ =1M and $|T_2|$=100k (times in secs).

		RDBMS A		RDBMS B		RDBMS C	
p	q	nulls	invalid	nulls	invalid	nulls	invalid
10	10	16.0	24.8	5.0	6.2	1.1	1.2
20	10	16.8	23.8	5.9	8.8	1.0	1.1
40	10	17.4	24.2	8.3	12.3	1.0	1.0
80	10	19.0	27.6	12.4	20.6	1.1	1.1
10	10	16.0	24.8	5.0	6.2	1.1	1.2
10	20	16.7	25.0	5.3	6.4	1.1	1.2
10	40	16.8	25.2	4.8	6.3	1.2	1.2
10	80	17.4	33.7	5.3	6.5	1.2	1.2

are opportunities for query optimization.

The previous experiments leave questions about skewed key distributions unanswered. Can we expect hash joins and sort-merge joins to be robust to skewed key distributions? Table V shows time growth inserting invalid keys with the four probabilistic distributions explained in Section 3. In this case T_1 does not have a secondary index on K. Recall from Table I when T_1 does not have a secondary index RDBMS A uses sort-merge join, whereas RDBMS B and RDBMS C use a hash join. We can observe skewed distributions do impact join performance for large tables (i.e., large $|\Delta|$), but do not matter for small Δ. We now analyze the widest gap (difference between minimum and maximum time) in each RDBMS for $|\Delta|$=1.6M. In RDBMS A the time for the zipf and constant distributions is twice the time for the uniform distribution. This is explained by the fact that the hash indexing mechanism incurs on many collisions for skewed distributions. In RDBMS B the pattern is different: times grow about 40% between the constant and the rest. This is explained by the fact that B-trees efficiently handle duplicate values during join. In RDBMS C the join algorithms are marginally influenced by the probabilistic distribution. However, both RDBMS B and RDBMS C use hash joins and they have different trends. We conclude that the probabilistic distribution impacts sort-merge and hash joins.

Table V. Impact of probabilistic distribution of key; $|T_2|$=100k (times in secs).

		RDBMS A				RDBMS B				RDBMS C			
$\|\Delta\|$	$\|T_1\|$	const	unif	zipf	geom	cons	unif	zipf	geom	const	unif	zipf	geom
0	1,000,000	13.3	13.3	13.3	13.3	3.6	3.6	3.6	3.6	0.5	0.6	0.5	0.5
100,000	1,100,000	20.2	20.2	19.5	19.2	4.4	6.8	5.5	5.4	0.8	0.7	0.8	0.7
200,000	1,200,000	24.5	27.5	24.5	28.2	5.1	5.1	6.1	6.1	1.0	1.0	1.0	0.9
400,000	1,400,000	33.1	27.8	35.6	32.1	5.5	5.7	6.3	5.7	0.9	0.9	1.0	0.9
800,000	1,800,000	37.4	28.8	36.5	33.0	4.8	5.5	6.4	5.9	1.2	1.2	1.1	1.2
1,600,000	2,600,000	59.4	31.5	62.3	43.5	5.1	7.0	6.8	6.7	1.1	1.2	1.1	1.1

Table VI. Query Optimization: view materialization Y/N (times in secs).

			RDBMS A		RDBMS B		RDBMS C	
$\|\Delta\|$	$\|T_1\|$	$\|T_2\|$	Y	N	Y	N	Y	N
0	1,000K	1k	13.2	13.2	2.4	2.4	0.5	0.5
100K	1,100K	1k	13.2	14.0	2.4	2.6	0.5	0.6
800K	1,800K	1k	13.2	14.8	2.4	2.8	0.5	1.5
1600K	2,600K	1k	13.2	14.9	2.4	3.4	0.5	1.7
0	1,000K	100k	14.1	14.4	3.6	4.0	0.5	0.5
100K	1,100K	100k	14.1	19.6	3.6	5.7	0.5	0.6
800K	1,800K	100k	14.1	24.8	3.6	6.7	0.5	1.4
1600K	2,600K	100k	14.1	31.5	3.6	7.0	0.5	1.7

4.5 Query Optimizations

Table VI compares the use of the materialized view, the first optimization. This query optimization produces a significant speedup in tables with many invalid keys in all RDBMSs. Time savings are proportional the number of invalid keys, as expected. All RDBMSs take advantage of the precomputation to avoid searching for invalid keys. The impact is more significant when T_2 is large.

We evaluate our second query optimization: creating a secondary index for joins, shown in Table VII. Since invalid keys take longer time to process than null keys they are the default. T_1 has initially 1M rows with valid key values in K and T_2 has 100k rows. We add increments, doubling increment size, to T_1 having nulls or uniformly distributed invalids in K. The index is created before measuring time and it is dropped right after measuring time. This procedure assured the index was created in an optimal manner avoiding insertion overhead. Time measurements exclude the time to create the index. Table VII shows the secondary index is very important for efficient processing of nulls and invalid keys in RDBMS A and RDBMS C. The speedup is significant when $|\Delta|$=1.6M. In contrast, the existence of a secondary index does not benefit RDBMS B which relies on hash joins. Experiments (not shown) with a smaller $|T_2|$=1k produce similar results. In RDBMS A and RDBMS C the secondary index was always exploited by the join algorithm. In fact, RDBMS C changed its join algorithm. It was interesting that in RDBMS B the query optimizer always favored a table scan over using the index. That is, the secondary index was not exploited. In the experiments shown here we forced RDBMS B to use the secondary index, producing a marginal speedup improvement in a few cases. That is, it helped little or it was worse as predicted by the query optimizer.

For the third optimization we study query evaluation time reordering joins on a 2-way join with tables S, T_1, T_2. We assume table S has two foreign keys K_1 and K_2 that refer to tables T_1 and T_2, respectively. The number of S rows with invalid keys for K_1

Table VII. Query Optimization: secondary index on foreign key (times in secs).

		RDBMS A		RDBMS B		RDBMS C	
Δ	$\|T_1\|$	Y	N	Y	N	Y	N
0	1,000,000	13.8	14.4	3.9	4.0	0.5	0.6
100K	1,100,000	13.7	19.6	5.5	5.8	0.5	0.7
800K	1,800,000	14.4	24.8	6.4	5.5	0.5	1.3
1,600K	2,600,000	15.3	31.5	7.3	7.0	0.5	1.2

Table VIII. Query Optimization: join reordering for 2-way join with 3 tables (times in secs).

		RDBMS A		RDBMS B		RDBMS C	
$\|K_1\|$	$\|K_2\|$	Y	N	Y	N	Y	N
1000k	0	42	72	17	39	2.2	6.8
1000k	100k	40	68	16	37	2.1	6.2
1000k	1000k	35	35	15	15	1.9	1.9

Table IX. Rank of importance of each parameter.

Parameter	RDBMS A	RDBMS B	RDBMS C
Secondary index	1	6	1
$\|\Delta\|$	2	2	2
$\|T_2\|$	3	1	3
K pdf	4	4	4
p	5	3	4
q	6	5	4

is $|K_1|$ and for K_2 it is $|K_2|$. Notice one table has two foreign keys, instead of having a FK "chain" from S to T_1 and then from T_1 to T_2. In such case, joins that produce smaller tables should get evaluated first. Table sizes are as follows: $|S|$=2M, $|T_1|$= 1M, $|T_2|$=100k. For this experiment we assume K_1 and K_2 have independent errors, uniformly distributed. We vary the number of invalid keys in K_2 to study the impact on time. Table VIII compares the use of the reordering optimization turning it on and off. As we can see join reordering produces a time decrease in all RDBMS. The speedup is more significant when one key (K_1) has many invalid values and the other key (K_2) has no invalid values.

4.6 Summary and Recommendations

Each RDBMS optimizer produced different query evaluation plans for a natural join between two tables. The number of rows with invalid keys did not influence the query plan. We found hash-joins were more efficient to process nulls/invalids than sort-merge join, but more extensive experiments are needed to understand trade-offs.

Table IX shows the rank of importance of each parameter we analyzed in the experiments. We base this ranking on the impact on time (as percentage of running time) by the given parameter. A large $|\Delta|$ implies a larger T_1 with a large fraction of rows with null and invalid key values. In general, the secondary index and the fraction of rows with invalid keys tend to be the dominating factors, in that order; the exception for the secondary index is RDBMS B, which favors hash joins.

In general, we recommend creating a secondary index for a foreign key column with invalid or null values that will be used frequently (second optimization). Creating materialized views (temporary tables) that exclude invalid keys in the referencing table can produce a significant speedup when there is a large fraction of rows with invalid keys (first optimization). If invalid keys are unlikely to be useful it is recommended to substitute them for nulls because this can produce an important speedup in join processing, especially for large referenced tables. Nullifying invalid keys with skewed distributions can help improving join performance or in the worst case leave it unaffected. Skewed distributions of invalid keys impact performance. On the other hand, row size and table size of referenced tables play a small role in performance.

5. RELATED WORK

The join operator is the most demanding operator in relational algebra. Therefore, there is a lot of work on efficient join processing in general [Garcia-Molina et al. 2001] and in OLAP processing. In [Mishra and Eich 1992] the authors present an extensive survey of different kinds of joins and various implementation techniques. References [Swami 1989; Tay 1990; Gelder 1993] and others study the optimization of the join operator on several tables. A particular work that studied join evaluation with nulls is [Yuan and Chiang 1998]; this work proposes an extended relational algebra to handle nulls, but does not consider the physical database aspects as we do. Removing or adding information in an incomplete database having nulls is studied in [Keller and Winslett 1984]; the focus of this paper is to repair the database based on some constraints, rather than dealing with joins with nulls. Re-optimization and iterative improvement is extensively studied for complex queries in [Ioannidis and Kang 1990; Kabra and DeWitt 1998]. Reference [Chaudhuri 1998] presents an overview of query optimization that includes interesting discussion on the join operator.

Our optimizations for join queries on incomplete databases are related to research on measuring and improving

data quality in relational databases considering referential integrity [García-García and Ordonez 2008; Ordonez and García-García 2008]. Other related work on incomplete databases and database integration includes the following. In [Hurson et al. 1987] the authors examine the relationship between incomplete information and the join operation. They present a join module for a database machine and a time analysis to evaluate its performance. In contrast, our study evaluates the join operator, from a query optimization point of view in databases with data quality issues. We should mention that most join evaluation techniques consider the case that many key values in T_2 will not be referenced, whereas we also consider that many T_1 values do not have valid references. In [Lim and Chiang 2000] the authors investigate the correct integration of relationship instances obtained from different source databases, focusing in detecting semantic conflicts and reconciling them. In summary, we have gone further by studying joins in the presence of referential integrity issues.

6. CONCLUSIONS

We analyzed the performance of natural join queries in three RDBMSs in the presence of null, invalid and valid keys. We studied three common relational query optimizations: view materialization excluding invalid keys, creation of a secondary index on a foreign key and join reordering based on number of invalid key values. We conducted an experimental evaluation on three RDBMSs. In all RDBMSs, the two alternative join algorithms were hash-joins and sort-merge joins, which were not influenced by the fraction of rows with invalid keys. Our findings were the following. There still exist efficiency issues when processing joins, highlighting opportunities for query optimization. The number of rows with invalid keys is more important when the referenced table is large. The time to process valid keys is greater than or equal to processing keys invalid values, highlighting the I/O cost to produce output rows. Skewed probability distributions for invalid keys indeed impact join performance on

large tables when there is a significant fraction of rows with nulls or invalid keys. Row size of the referencing table is far more important than the row size of the referenced table. However, row size is a minor I/O factor, compared to other factors. Some query optimizations significantly improved join performance in tables with a large number of invalid keys. View materialization produced a significant speedup improvement in every case since it avoided searching invalid values completely. The secondary index produced a significant speedup, except in one RDBMS, which favored hash joins in every plan. Join reordering proved best when one foreign key had many invalid values and the other foreign key had a very low number of invalid values. Finally, view materialization proved to be a widely applicable optimization because it worked well in all RDBMSs.

Even though optimizing join queries is an old topic, there are interesting research issues. We intend to study multi-way joins with several foreign keys in more depth, especially for chained foreign key relationships. We would like to better understand if hash joins are more efficient than sort-merge joins for tables with many invalid values or having skewed key distributions. Efficiently handling invalid keys when there are queries combining joins and aggregations is another important problem. We want to study other natural join queries, where two tables share the same primary key (e.g., diffierent summarizations from same normalized table), or two tables share the same foreign key (i.e., the foreign key does not act as a primary key in either table). Join queries with composite keys, where subsets of attributes may be invalid, is a more general aspect that also needs to be analyzed.

ACKNOWLEDGMENTS

We would like to thank the help from Zhibo Chen, Sasi K. Pitchaimalai and Santiago Suarez-Castañon to run some experiments. The first author was supported by National Science Foundation grants CCF 0937562 and IIS 0914861.

REFERENCES

CHAUDHURI, S. 1998. An overview of query optimization in relational systems. In *Proc. ACM PODS Conference*. 84–93.

CODD, E. 1979. Extending the database relational model to capture more meaning. *ACM TODS 4*, 4, 397–434.

ELMASRI, R. AND NAVATHE, S. B. 2000. *Fundamentals of Database Systems*, 3rd ed. Addison/ Wesley, Redwood City, California.

GARCÍA-GARCÍA, J. AND ORDONEZ, C. 2008. Estimating and bounding aggregations in databases with referential integrity errors. In *Proc. ACM DOLAP Workshop*. 49–56.

GARCIA-MOLINA, H., ULLMAN, J., AND WIDOM, J. 2001. *Database Systems: The Complete Book*, 1st ed. Prentice Hall.

GELDER, A. V. 1993. Multiple join size estimation by virtual domains (extended abstract). In *Proc. ACM PODS Conference*. 180–189.

HURSON, A. R., MILLER, L. L., AND PAKZAD, S. H. 1987. Incomplete information and the join operation in database machines. In *ACM '87: Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow*. IEEE Computer Society Press, Los Alamitos, CA, USA, 436–443.

IOANNIDIS, Y. E. AND KANG, Y. 1990. Randomized algorithms for optimizing large join queries. In *Proc. ACM SIGMOD Conference*. 312–321.

KABRA, N. AND DEWITT, D. J. 1998. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. ACM SIGMOD Conference.* 106–117.

KELLER, A. AND WINSLETT, M. 1984. Approaches for updating databases with incomplete information and nulls. In *Proc. IEEE ICDE Conference.*

LIM, E. AND CHIANG, R. 2000. The integration of relationship instances from heterogeneous databases. *Decis. Support Syst. 29-2*, 3-4, 153–167.

MISHRA, P. AND EICH, M. H. 1992. Join processing in relational databases. *ACM Comput. Surv. 24*, 1, 63–113.

ORDONEZ, C. AND GARCÍA-GARCÍA, J. 2008. Referential integrity quality metrics. *Decision Support Systems Journal 44*, 2, 495–508.

SWAMI, A. 1989. Optimization of large join queries: combining heuristics and combinatorial techniques. In *Proc. ACM SIGMOD Conference.* 367–376.

TAY, Y. C. 1990. On the optimality of strategies for multiple joins. In *Proc. ACM PODS Conference.* 124–131.

YUAN, L. AND CHIANG, D. 1998. A sound and complete query evaluation algorithm for relational databases with null values. In *Proc. ACM SIGMOD Conference.*

Carlos Ordonez got a Ph.D. degree in Computer Science from the Georgia Institute of Technology, USA, in 2000. Dr Ordonez worked six years extending the Teradata DBMS with advanced data mining techniques to analyze large databases. Dr. Ordonez is currently an Assistant Professor at the University of Houston. His research is centered on the integration of machine learning and statistical techniques into database systems to analyze large data sets as well as their application to scientific problems. Dr Ordonez research has produced over 60 papers, over 500 citations and has been funded by NSF.

Javier García-García received a PhD degree in computer science from UNAM University, Mexico. Dr García-García worked managing IT departments at several public and private institutions in Mexico for more than 20 years. Currently, he is a professor at the UNAM University, where he conducts research on relational database systems, focusing on data quality and data mining. He is a member of ACM and IEEE.