Remote Comparison of Database Tables
Technical Report A/375/CRI

Fabien Coelho
fabien.coelho@ensmp.fr
Centre de Recherche en Informatique, École des mines de Paris,
35, rue Saint-Honoré, 77305 Fontainebleau, France.

February 6, 2006

Abstract

Database systems hold mission critical information in all organizations. Data are often replicated for being processed by different applications as well as for disaster recovery. In order to help handle these replications, it is useful to compare remote sets of data to detect unwanted changes due to hardware, system, software, application, communication or human errors... This paper presents an algorithm based on operations and functions available in all relational database systems to reconciliate remote tables by identifying inserted, updated or deleted tuples with a small amount of communication. A tree of checksums which covers the table contents is computed on each side and merged level by level to identify the differing keys. The algorithm is somehow an adaptation of Metzner [9] to our particular context. A prototype implementation [4] is available as a free software. Experiments show our approach to be effective even for tables available on a fast local network.

1 Introduction

Relational database systems must hold reliably mission critical information in all organizations. These data are often stored in multiple instances through synchronous or asynchronous replication tools or with bulk data transfers dedicated to various transactional and decisional applications. These data replications can help enhance load sharing, handle system failures, application, software or hardware migrations, as well as applicative transfers from one site to another.

As trust does not preclude control, it is often desired to compare the data between remote systems and identify inserted, deleted or updated tuples. This is known as the set reconciliation problem. Few differences are expected between both data sets. Key issues include big data volumes, site remoteness and low bandwidth. Transferring the whole data for comparison is not a realistic option.

This paper presents a portable algorithm to compare relational database tables which may reside on remote and heterogeneous DBMS such as open source PostgreSQL and MySQL or proprietary Oracle and DB2. The algorithm finds the key of inserted, updated or deleted tuples with respect to a parametric subset of rows and columns. It relies on simple SQL constructs and functions available in any relational database system. Summaries are extracted on each server and transfered to the client system where the reconciliation is performed. A block parameter allows to optimize the latency/bandwidth tradeoff. This algorithm is somehow adapted from Metzner [9] with a parametric group size and a special handling of tuple keys.

The paper is organized as follows: after this introduction, Section 2 details the related work. Next, Section 3 presents the comparison algorithm and the SQL queries performed to build the necessary checksum structures and compute the differences. The algorithm is then analyzed in Section 4 and discussed in Section 5. Section 6 describes our prototype implementation [4] developed in perl, a fully portable scripting language. Experiments are reported in Section 7. Finally, Section 8 concludes our presentation.

2 Related Work

Suel and Nemon overview paper [13] analyzes many comparison algorithms for delta compression and remote synchronization. A first class of problem addresses locally available data sets, and targets identifying deltas. A second class deal with data stored on remote locations, and aims at identifying missing or differing parts without actually transferring the data. Another key issue is whether the data are naturally ordered, such as a string or a file composed of pages, or considered as a set of distinct unrelated elements, such as the files in a file system or the tuples of a relation.

String to string combinatorial research problems are described in 1972 [3]. Solutions are found for string correction [18], string searching [7, 2], and delta text or binary file generation [12, 10, 16] such as performed by the `diff` command. These various approaches do not apply to our problem as they deal with locally available data sequences.

The remote comparison problem has been addressed with various techniques. Metzner [9] presents a checksum binary-tree algorithm which identifies differing pages of a file. Our approach is somehow an adaptation to the relation set structure with a special management of the keys to detect tuple updates as opposed to inserts and deletes, and a block parameter to adjust the tree degree.

The practical `rsync` algorithm [15, 14] is well known to system administrators. It is asymmetrical in nature. Blocks of data already available on one side are identified and complementary missing data are sent to the other side. Block shifts are identified at the byte level thanks to a sliding checksum computation. With respect to our problem, such approach could result in easy identification of inserts, but very poor network performances for updates and deletes.

Elegant coding-theory based solutions [1, 6, 11] reduce the number of communication rounds and the amount of transfers for comparing remote data sets. The key idea is that as the data are already available with very few differences, only the error correcting part of a virtual transmission is sent and allows to reconstruct the differences. However, such techniques need significant mathematical computations on both sides which are not available with the standard SQL functions of relational database system.

Maxia [8] presents several asymmetric algorithms to compare remote relations using checksums. One level of summary table is used, leading to an overall communication complexity in $\mathcal{O}(k(b + n/b))$ where k, n, b are the number of differences, the table size and a block size. The algorithm for inserts and updates does not detect deletes and does not perform properly in some limit cases, whereas the algorithm for deletes does not work if other operations where performed. In contrast, our solution provides a single symmetric algorithm for detecting all differences with a better communication complexity.

Software products are also available for purchase such as DBDiff [5]. Although this tool compares table contents, the actual algorithm used and its bandwidth requirements are unclear. Such package also focus on table structure comparisons, so as to derive SQL ALTER commands necessary to shift from one relational schema to another.

3 Remote Comparison Algorithm

Let us now present the hierarchical algorithm for comparing two remote database tables named T with K the primary key, V the attributes to be compared, and W a condition to select a subset of the rows to be analyzed.

Let n be the table size (number of rows in T), l the tuple length (size of attributes), k the number of differences to be found, h a checksum function (possibly a cryptographic hash function) of size c bits, b a power of two block size used as a folding factor. Let $d = \lceil \frac{\ln(n)}{\ln(b)} \rceil$ be the tree depth, and $m_i = b^{d-i} - 1$ the grouping mask for level i, plus $m_{d+1} = 0$.

The algorithm is fully symmetrical. It computes on both sides a hierarchical tree of summaries shown in Fig. 1. They are then scanned from the root downwards to identify the differing tuple keys by investigating the differences concurrently. The reconciliation is achieved by merging the sum-

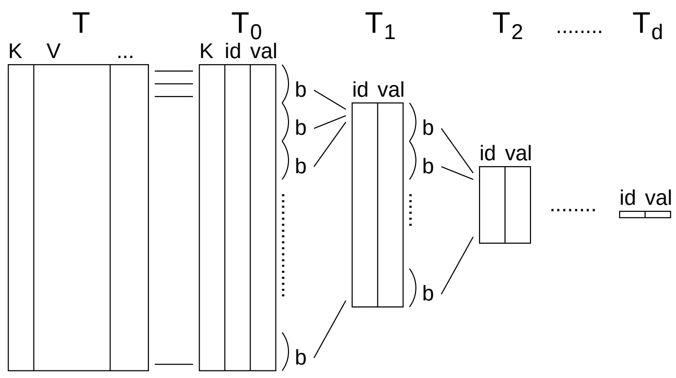

Figure 1: Full tree of summary tables

maries at each level. It does not decide what to do with the differences, but simply locates the offending keys and reports them.

First, an initial summary table T_0 storing both tuple keys and signatures is built as shown in Fig. 2. Second, aggregations in Fig. 3 computes the summary tree structure for all the tuples. The last table only holds one summary which checksums the whole table. Third, remote selects in Fig. 4 on the summary tables allows to reconciliate the tuples and thus to identify inserts, updates and deletes by a merge algorithm which deals with key and value checksums in Fig 5. It first compares the one row of table T_d. If they are different, it proceeds down the tree to check for the source of the differences up to the lower checksum table where the actual tuple keys are available.

4 Analysis

The number of requests of the client-server protocol depends whether there are differences: an initial request gets the table size which is necessary to compute the tree depth, and then from 1 up to $d+1$ queries are performed for the actual reconciliation. Thus there are $\mathcal{O}(\ln(n)/\ln(b))$ requests.

The amount of data communicated at each stage depends on the selected block size and the number of differences to be found. For small k, $\mathcal{O}(kcb\lceil\ln(n)/\ln(b)\rceil)$ data are communicated: each difference is investigated on the depth of the tree, and each found block to be merged contains cb bits. As k grows, a steady state is reached when all blocks at level 0 are scanned as they all contain at least one difference: the communication is then $\mathcal{O}(cnb/(b-1))$.

There is a latency/bandwidth tradeoff implied by the choice of the block folding factor: the higher b the higher the amount of data to be transfered, but the lower the number of requests as the depth is reduced.

The amount of computation and I/O performed by the database depends

```
CREATE TEMP TABLE $T_0$ AS
SELECT
  $K$ AS key,
  $h(K)$ AS id,
  $h(K, V)$ AS val
FROM $T$ WHERE $W$;
```

Figure 2: Initial summary table

```
CREATE TEMP TABLE $T_i$ AS
SELECT
  id&$m_i$ AS id,
  XOR(val) AS val
FROM $T_{i-1}$
GROUP BY id&$m_i$;
```

Figure 3: Summary tables

```
// get checksums at level i
list getIds(c, i, what)
  withkey = (i==0)? ", key": ""
  return sql2list(c, "
    SELECT id, val $withkey
    FROM $T_i$
    WHERE id&$m_{i+1}$ IN ($what)
    ORDER BY id ASC")

// show a block of matching keys
showKeys(c, msg, l)
  for v,i in l
    for key in sql2list(c, "
      SELECT key FROM $T_0$
      WHERE id&$m_i$ = $v")
      print "$msg $k"
```

Figure 4: Summary queries

```
// reconciliate on connections c1 c2
merge(c1, c2)
  list curr = (0), next, ldel, lins;
  level=d;
  while (level>= 0 and curr)
    // get checksums at level
    list lid1 = GetIds(c1, level, curr),
    list lid2 = GetIds(c2, level, curr);
    // merge both sorted lists
    while (i1 or i2 or lid1 or lid2)
      i1,v1,k1 = shift(lid1) if no i1;
      i2,v2,k2 = shift(lid2) if no i2;
      if (i1 and i2 and i1==i2)
        // matching key checksum
        if (v1!=v2)
          // differing value checksum
          if (level==0) print "UPDATE $k1"
          else append i1 to next
        elsif (no i2 or i1<i2)
          // single checksum in lid1
          if (level==0) print "DELETE $k1"
          else append (i1,level) to ldel
          undef i1
        elsif (no i1 or i1>i2)
          // single checksum in lid2
          if (level==0) print "INSERT $k2"
          else append (i2,level) to lins
          undef i2
    level--
    curr = next
  // whole block differences
  showKeys(c2, "INSERT",lins)
  showKeys(c1, "DELETE",ldel)
```

Figure 5: Reconciliation merge algorithm

on the optimizations implemented by the query processor and on the data size. For building the initial summary table, all data $\mathcal{O}(nl)$ must be read. The first aggregation can be performed with a merge technique which requires a sort in $\mathcal{O}(n \ln(n))$, or an hash technique which is done on the fly in $\mathcal{O}(n)$, and the subsequent ones are reduced by power of b leading to a $b/(b-1)$ factor. The final requests for the merge phase just require to scan some data, which may depends on the presence of relevant indexes to be created possibly in $n \ln(n)$ operations. Thus the overall computation cost on each side is $\mathcal{O}(nl + n \ln(n) b/(b-1))$.

5 Discussion

A key hypothesis is that few differences are expected: otherwise the search process scans most of the table through many requests, although an ordered scan of the initial table would allow to identify the missing or differing items in a single pass. If the hypothesis is not met, the implementation may allow the user to stop the computation when the number of differences encountered is above a given threshold.

One checksum computation is performed on the key and another on the key and value attributes. The first hash aims at randomizing the key distribution so that aggregations group tuples evenly, and so that the computations do not depend of the key type and composition. It is also needed to differentiate updates from inserts and deletes. The second hash of the key and value part identifies the items. The key in the second checksum is necessary and is not redundant with the previous hash: otherwise a simple exchange of values between two tuples would not be detected if they are aggregated in the same group.

As is usual with the choice of the checksum functions, its size (c bits) and quality should be good enough to avoid collisions. The consequence of collisions in the key part is that two tuples are not differentiated, hence a difference detected on one would also be reported about the other as a false positive. A collision of the key and value hash for the same key hash would result in an actual difference not to be reported, thus leading to a more annoying false negative. This later case is rare as it is conditional to the collision occurring for the very same key hash, that is either with the same key or with a collision of the key hash part: indeed, the second hash is only used for matching key hashes in the merge procedure. Cryptographic hash functions with $c >= 128$ make collisions quite improbable.

The tree of aggregations computes a common checksum by combining tuple chunks. The operation used should treat individual checksum bits equally. Exclusive-or (XOR) is the usual operator of choice if available. If not, the SUM aggregation can be considered, provided it applies to the checksum result type. The group criterion should also be compatible with the checksum result and allow to define tuple chunks. In order to be able to compute directly the group of a tuple at any level, its computation must only depend on the level and not on the groups computed at the preceding levels. This property is achieved by a binary mask or a modulo operations on the power of an integer.

The algorithm is fully symmetrical. Inserts and deletes are only parted on the convention that the second table serves as the reference in the comparison, but the structure of the algorithm is the same on both part. The algorithm handles missing intermediate keys with two special lists, `lins` and `ldel`. This case arises if a whole chunk of tuples is removed or added.

As noted by Maxia [8], it is possible to maintain the checksum directly in the initial table or in another by mean of trigger procedures, so that they would not need to be computed over and over. It could also be integrated in the database as a new kind of hash index dedicated to remote table comparison.

6 Implementation

A prototype implementation which targets PostgreSQL [17] is available [4]. The tool is implemented as a stand-alone perl [19] script which connects to the databases through the standard DBI interface. Several functionalities are needed in order to implement the scheme: the ability to replace null values, a checksum function, a grouping criteria and a relevant aggregate function.

First, as NULL values propagate through SQL functions, they must be dealt with in order to keep meaningful checksums. PostgreSQL provides the `COALESCE` function which can be used to substitute NULL values. Second, PostgreSQL includes the `MD5` cryptographic hash function on text or byte-array data, which although not cryptographic secure any more [20] is quite good enough as a checksum. Third, a criterion must be chosen to group the tuples in the tree building phase. It must be compatible with the data type holding the key checksum. For integer types, modulo operations can be used to handle any folding factor. For binary types, a mask comparison provides a simple solution with power of two block sizes. Finally, a checksum combining aggregate function must be provided. Standard SQL provides COUNT, SUM, AVG and STDDEV aggregates which do not optimally suit our needs. However, adding an aggregate XOR based on the corresponding binary operator is simple, as shown in Fig. 6.

There is a type issue in the above simple approach based on standard function available in current PostgreSQL: the `MD5` function result is an hexadecimal text. However, no exclusive-or is available on such a format, and there is no standard way to convert this format to a compatible format. A possible solution is to decode the hexadecimal string into a byte array, and

```
CREATE AGGREGATE XOR(
  BASETYPE = BIT,
  SFUNC = bitxor,
  STYPE = BIT
);
```

Figure 6: Add XOR aggregate

```
LOAD '$libdir/casts';
CREATE OR REPLACE
FUNCTION bytea2varbit(BYTEA)
RETURNS VARBIT
LANGUAGE C
AS '$libdir/casts', 'bytea2varbit';
```

Figure 7: Add a dynamically linked function

to rely on loadable user defined function in Fig. 7 to convert this type to the bit array type for which the XOR operator is implemented.

The table comparison can be restricted so as to perform partial checks. The implementation specifies the attributes to be considered with an option, and the rows can be selected with a parametric WHERE close.

As far as portability is concerned, our implementation targets postgreSQL, but can be easily tuned for other databases through parametric templates. The generated SQL queries are pretty straightforward. The main portability issue is rather the availability of the support functions and operators. Although the checksum MD5 function is widely available, the XOR aggregate is not, and some database systems may not be extensible. In such cases, the SUM aggregation can be used as an alternative.

7 Experiments

Let us report experimental results obtained with our tool. Randomly generated tables from 10K to 500K rows with 400-byte long records are compared in a bandwidth bound environment. Varying block sizes and number of differences are investigated.

The following measures are to be taken with caution: they are sensitive to many parameters such as the hardware including hard disk, cpu and memory, database configuration and optimization, as well as network latency, bandwidth, mtu and congestion status... Overall elapsed times were used. It encompasses both computation and network, the preeminence of which varies depending on the actual conditions.

Fig. 8 shows data collected with connections established on a local area network, without any bandwidth restriction. The horizontal axis is the $\log_2$ group size used for building the checksum tree. The vertical axis in the normalized time to perform the reconciliation in ms per tuple. Different table sizes are investigated to recover 3 differences. All figures are very similar whatever the table and group size, but for the smallest table: the algorithm performs close to linearly in this context, where the dominating factor is the database computations.

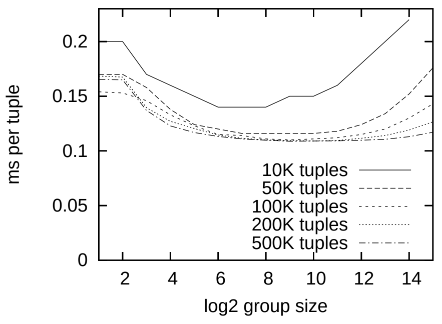

Figure 8: Comp. time per tuple, LAN

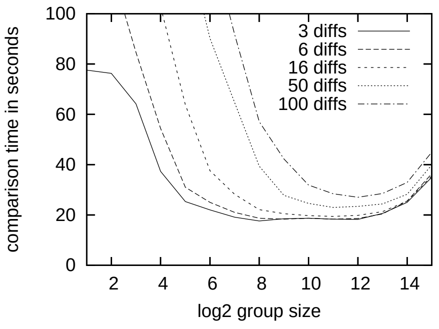

Figure 9: Comp. time vs block, avg bw

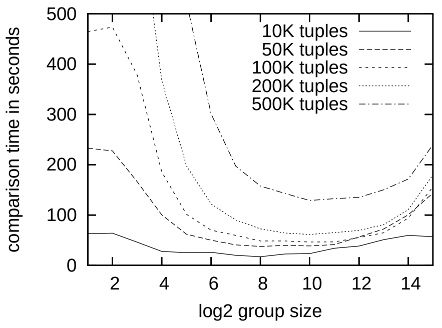

Figure 10: Comp. time vs block, low bw

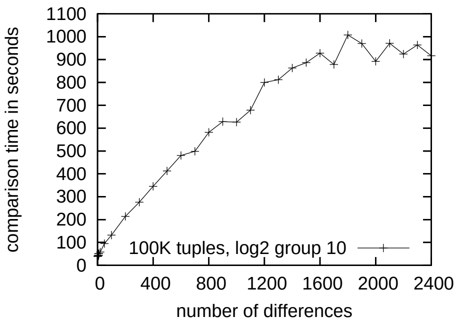

Figure 11: Comp. time vs diff, low bw

In Fig. 9, an average less than 100KB/s network link is used to reconciliate 100K tuple tables with different block parameters. The number of differing tuples is made to vary from 3 to 100, and the comparison time is displayed in seconds. Average block from $1024 = 2^{10}$ to $8192 = 2^{13}$ performs best. Small block values are loaded by network latency and congestion.

A low bandwidth (modem class) network under partial congestion is used in Figures 10 and 11. In the former, 3 differences are investigated for 5 table sizes and varying block size. The network bandwidth and latency dominates the measures: for average block sizes, the measures are quite independent of the table size. In the later figure, the block size is fixed to 10 and the number of differences is scaled up. The linear dependency of the algorithm in network bound context clearly shows for large number of differences, and for higher figures the saturation effect is encountered when all checksum blocks are fetched.

These figures can be compared to the direct approach which consist of downloading the data from one site to the other, restoring them into a database and then compare them with an external join. For the 100K table size and low bandwidth, the download requires one hour. For the average bandwidth solution, the needed time is 10 minutes, and the LAN time is about 10 seconds. Our algorithm implementation performs better in all our tests that this brute force approach, even with the high bandwidth local area network. Based on our experiments, the overall best block size for our algorithm in widely differing network conditions is in the 1024-4096 range.

8 Conclusion

This paper describes an adaptation of Metzner [9] algorithm dedicated to remote relational database table comparisons with a parametric block size. It allows to identify the key of inserted, deleted or updated tuples. This algorithm can be implemented on top of reasonable instances of SQL: most of the checksum work is performed through database requests, and the client tool basically performs a reconciliation merge of partial checksums fetched level by level.

All the experiments of our remote comparison algorithm show better performances than the brute force download solution, whatever the chosen parameters. Although this is not a general rule, this shows nevertheless that our implementation provides an elegant and portable solution to remote comparison of database tables even on a high bandwidth network. Remote comparison really mean not directly available in the same database.

Future work includes improving the implementation so as to deal with heterogeneous database systems without additional parameterization.

References

[1] K. A. S. Abdel-Ghaffar and A. El Abbadi. An optimal strategy for comparing file copies. *IEEE Trans. Parallel Distrib. Syst.*, 5(1):87–93, 1994.

[2] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[3] Vaclav Chvatal, David A. Klarner, and Donald E. Knuth. Selected combinatorial research problems. Technical report, Stanford University, Stanford, CA, USA, 1972.

[4] Fabien Coelho. Pg Comparator. Software available on `http://pgfoundry.org/projects/pg-comparator`, August 2004.

[5] DKG Advanced Solutions. DBDiff for windows. `http://www.dkgas.com/`, 2004.

[6] M.G. Karpovsky, L.B. Levitin, and A.; Trachtenberg. Data verification and reconciliation with generalized error-control codes. *IEEE Transactions on Information Theory*, 49(7):1788–1793, July 2003.

[7] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, June 1977.

[8] Giuseppe Maxia. Taming the distributed database problem: A case study using MySQL. *Sys Admin*, 13(8):29–40, August 2004.

[9] John J. Metzner. A parity structure for large remotely located replicated data files. *IEEE Trans. Computers*, 32(8):727–730, 1983.

[10] Webb Miller and Eugene W. Myers. A file comparison program. *Software–Practice and Experience*, 15(11):1025–1040, 1985.

[11] Y. Minsky, A. Trachtenberg, and R. Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*, 49(9):2213–2218, September 2003. Also Proceedings of ISIT'2001.

[12] Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[13] T. Suel and N. Memon. *Lossless Compression Handbook*, chapter Algorithms for Delta Compression and Remote File Synchronization. Academic Press, 2002.

[14] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization.* PhD thesis, Australian National University, 1999.

[15] Andrew Tridgell and P. MacKerras. The rsync algorithm. TR-CS 96-05, Australian National University, June 1996.

[16] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.

[17] Various Contributors. PostgreSQL free open source database. `http://www.postgresql.org/`, 1996.

[18] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[19] Larry Wall. Perl scripting language. `http://www.perl.org/`, 1987.

[20] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199, 2004. `http://eprint.iacr.org/`, CRYPTO 2004 rump session.

Typeset with LaTeX, document revision 333