Remote Comparison of Database Tables

Fabien Coelho CRI, Maths & Systems, MINES ParisTech,
35 rue Saint Honoré, 77305 Fontainebleau cedex, France.
fabien.coelho@mines-paristech.fr

***Abstract*—Database systems hold mission critical data in all organizations. These data are often replicated for being processed by different applications as well as for disaster recovery. In order to help handle these replications, remote sets of data must be compared to detect unwanted changes due to hardware, system, software, application, communication or human errors. We present an algorithm based on operations and functions already available in relational database systems to reconcile 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. A prototype implementation is available as a free software. Experiments show our approach to be effective even for tables available on a local network. This algorithm provides a communication efficient and general solution for comparing remote database tables.**

***Keywords*—remote set reconciliation; data replication.**

I. 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 desirable to compare the data between remote systems and identify inserted, deleted or updated tuples, to detect errors or to resynchronize data. 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 under these assumptions.

This paper presents a generic 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. We assume that the compared tables are composed of records identified by a key, say the primary key of the relation. We do not make other assumptions on the data sets, such as the availability of attributes that may be used for grouping data, or restrict the number, type, size or value of attributes involved in the comparison either as the key or as the other columns. The algorithm finds the key of inserted, updated or deleted tuples with respect to a parametric subset of rows (part of the records are investigated) and columns (the comparison is restricted to some attributes). It relies on simple SQL constructs and functions available on all systems. Summaries are extracted on each server and transferred to the client system where the reconciliation is performed. A block parameter allows to optimize the latency/bandwidth trade-off.

The paper is organized as follows. Section II details the related work. Then, Section III presents the comparison algorithm and the SQL queries performed to build the necessary checksum tree and compute the differences. The algorithm is then analyzed in Section IV and discussed in Section V. Section VI describes our implementation. Experiments are reported in Section VII. Finally, Section VIII concludes our presentation.

II. Related Work

Suel and Nemon [1] analyze many comparison algorithms for delta compression and remote synchronization. A first class of problem addresses locally available data sets, and targets identifying deltas. Chvatal *et al.* [2] described string to string combinatorial research problems in 1972. Solutions are found for many of these problems, but these approaches do not apply to our problem as they deal with locally available data sequences. A second class deal with data stored on remote locations, and aims at identifying missing or differing parts without actually transferring the data. The remote set comparison problem has been addressed with various techniques. A 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, *i.e.*, unordered data, such as the files in a file system or the tuples of a relation.

Metzner [3] introduced a binary hash tree reconciliation to compare file contents. Our approach can be seen as an extension to handle tuple keys that identify records as opposed to the intrinsic page numbering that come with a file, and we use a parametric group size to select better trade-off opportunities depending on multiple optimization factors such as bandwidth, number of requests or disk I/Os. The practical `rsync` algorithm [4][5] 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.

Coding-theory based solutions [6][7][8] 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. Such techniques need significant mathematical computations on both sides that are not readily available with the standard SQL functions of relational database systems.

Descending recursive searches based on hashes over subsets on both sides were proposed [9], which is similar to [10] discussed later. Bloom filters are also used to statistically reduce the amount of data to communicate [11] in a synchronization, however this structure results in more false positive especially when the number of differences in the reconciliation is small. The same paper also briefly alludes to a Merkle tree [12], similar to Metzner and to our approach, for computing set differences.

In all of the above remote set comparison techniques, the differing elements are extracted either as inserts or deletes, but updates are not detected as such: as there is no concept of key to identify elements, updates cost two searches in the complexity computations to be identified as an insert and a delete. A few papers address our problem from the same viewpoint of (1) having a key to identify elements, (2) distinguishing updates from inserts and deletes and (3) being able to perform the algorithm through simple SQL queries.

Maxia [13] presents several asymmetric algorithms to compare remote tables 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 operate properly in some limit cases, whereas the algorithm for deletes does not work if other operations where performed. Ideally, a solution should detect all kind of differences with a low communication complexity. This will be our focus.

Schwartz [14][15] presents a top-down checksum approach to detect and update out-of-sync replication nodes. The algorithm uses the DBA knowledge in order to group tuple checksums according to attribute values, and thus may take advantage of existing indexes, but providing such a beneficial information is described as tricky by the author.

Vandiver [10] also presents a top-down *N-round-X-rows-hash* reconciliation algorithm for remote databases. The checksum approach is similar to ours, with the difference that the hash tree is not materialized, and by relying on an existing integer primary key. The first round computes hashes for records grouped in buckets based on the primary key value. Each differing bucket hash is then investigated by subdividing it, up to the N-th round where individual rows are considered. The algorithm can take advantage of existing indexes in the primary key, but it also requires some tuning. It performs best when defects are highly correlated.

Finally, software products are also available to identify differing rows, such as DBDiff [16] or DB Comparer [17]. Although these tools compare table contents, the actual algorithm used and its bandwidth requirements are unclear. Snapshots suggest a graphical user interface, which displays differing data to help the user perform a manual reconciliation. Such packages also focus on table structure comparisons, so as to derive SQL ALTER commands necessary to shift from one relational schema to another.

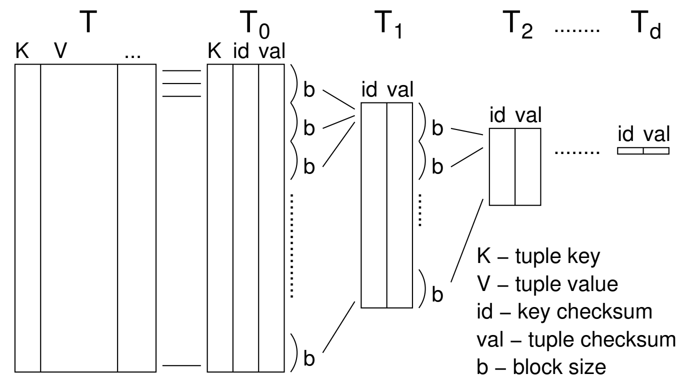

Fig. 1. Initial table T, checksum table T_0 and tree of summary tables $T_{i,i\geq1}$

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

Fig. 2. Checksum table T_0 with hash h

```
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;
```

Fig. 3. Summary tables $T_{i,i\geq1}$

III. 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.

The algorithm is fully symmetrical. It computes on both sides a hierarchical tree of summaries shown in Figure 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 summaries at each level. It does not decide what to do with the differences, but simply locates the offending keys and reports them. A natural continuation may be to transfer the offending data so as to re-synchronize the tables.

First, a checksum table T_0 storing both tuple keys and signatures is built as shown in Figure 2, using hash function h. Second, aggregations in Figure 3 compute the summary tree for all the tuples through a set of reduction masks $m_{i,i\geq1}$. The last table, T_d, only holds one summary checksum for the whole table. Third, remote selects in Figure 4 on the summary tables allows to reconcile the tuples and thus to identify inserts, updates and deletes by a merge algorithm that deals with key and value checksums in Figure 6. It first compares the one row of table T_d. If they are different, it walks through the tree to check for the source of the differences up to the checksum

```
// 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 $withkey")
```

Fig. 4. Summary extraction query

```
// 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"
```

Fig. 5. Key extraction query

```
// reconcile 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 "INSERT $k1"
          else append (i1,level) to lins
          undef i1
      elsif (no i1 or i1>i2)
          // single checksum in lid2
          if (level==0) print "DELETE $k2"
          else append (i2,level) to ldel
          undef i2
    level--
    curr = next
  // whole block differences
  showKeys(c1, "INSERT",lins)
  showKeys(c2, "DELETE",ldel)
```

Fig. 6. Reconciliation merge algorithm

table where the actual tuple keys are available. If a whole checksum block is empty on one side, the corresponding keys are extracted with a special query outlined in Figure 5.

The set of mask used in the aggregation is built as follows: Let n be the table size (number of rows in T), h a checksum function, possibly cryptographic. f the folding factor logarithm (that is the block size for folding is $b = 2^f$). Let $\ell = \lceil \ln(n)/\ln(2) \rceil$ be the closest power of 2 above the table size, $d = \lceil \ln(n)/\ln(b) \rceil$ be the tree depth, then $m_i = 2^\ell/b^i - 1 = 2^{\ell - if} - 1$ (when $1 \leq i \leq d$) is the grouping mask for level i, with $m_d = 0$.

It is important that m_1 should reduce the size of the first table by the folding factor, otherwise some folding factor values result in bad performances because the first folded table T_1 is not folded and takes as much time to compute as checksum table T_0. This is pointed out in Figure 7-3 of [10]. The last folding may be less efficient, but it is negligeable as the data volume involved is very small at the root of the tree.

IV. Analysis

Let us analyze the above algorithm with respect to the disk I/Os, computations and communications involved. We will use the following additional notations: l the tuple length (size of attributes), k the number of differences to be found, c the number of bits of checksum function h.

The amount of computation and I/O performed by the database depends on the optimizations implemented by the query processor and on the data size. The main cost is incurred in building the initial summary table, all $\mathcal{O}(nl)$ data of the initial data must be read, and $\mathcal{O}(nc)$ data must be written for the checksum table. Moreover, heavy computations are involved at this stage as two checksums are computed on the data read, which may also involve various data conversions depending on the actual data types. These checksums can be maintained as additional columns with triggers so that they would be available directly. Then, the first aggregation can be performed with a merge sort in $\mathcal{O}(n \ln(n))$, or an hash technique done on the fly in $\mathcal{O}(n)$, and the subsequent ones are reduced by power of b leading to an overall $b/(b-1)$ factor. The final requests for the merge phase just require to scan some data, which may be helped by indexes to be created possibly in $n \ln(n)$ operations. Thus, the overall computation cost on each side is $\mathcal{O}(nl + n\ln(n) \cdot b/(b-1))$.

The number of requests of the client-server protocol depends whether there are differences: An initial request gets the table size necessary to compute the tree depth and the relevant masks; Then, one batch of queries can build the checksum, summary tables, and return the initial root summary checksum. If they match, the tables are equals, otherwise the reconciliation must dig into the tree up to the leaves. Thus there are up to $\mathcal{O}(\ln(n)/\ln(b))$ 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, $\mathcal{O}(\lceil \ln(n)/\ln(b) \rceil \ln(n))kcb$ data are communicated: each differing id of size $\ln(n)$ 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 $\mathcal{O}(cnb/(b-1))$. This steady state comes around $k = n/2^f$. Such a saturation effect is encountered in the third set of experiments presented in Section VII.

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

V. Discussion

Our algorithm reduces the amount of data to be communicated through two key points: First, the attributes are compacted together with a checksum, which will usually be smaller than the data it summarizes. A checksum collision at this level would result in differences not to be found. Second, rows are aggregated together by building a summary tree, and only some of the computed records will be needed for the comparison. There again, a collision of the computed values at any level of the summary tree could result in differences not being found.

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.

Copyright (c) IARIA, 2011 ISBN:978-1-61208-115-1

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 on the key type and composition. It is also needed to differentiate updates from inserts and deletes. A simple integer evenly distributed key may be used instead if available. 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: if not included, a simple exchange of values between two tuples would not be detected if they are aggregated in the same group.

As usual with checksum functions, their size (c bits) and quality should be good enough to avoid collisions. A collision on the key hash in the checksum table makes two tuples identified as one, thus a difference detected on one would be reported about the other as a false positive, but this is easily filtered out by checking the tuple key also available in the checksum table. A collision of the value hash of two tuples in the checksum table does not have an impact on the search because tuples are also identified by their key hash. A collision of *both* the value hash for the same key hash on the checksum table would result in a difference not to be reported, thus leading to a false negative. A collision of the value hash in the summary tables for the same level and group would result in differences within these tables to be ignored. Cryptographic hash functions with $c \geq 128$ make such collisions improbable.

The tree of aggregations computes a common checksum by combining tuple chunks. This operation 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. Note that the central limit theorem is not an issue for the SUM aggregation: the more deterministic bit values are confined to upper bits of the sum, possibly removed by modulo arithmetic, while the lower bits only are significant in the hash combinations, and those stay as random as the inputs. The group criterion should also be compatible with the checksum result and allow to define tuple chunks. In order 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 first table serves as the reference in the comparison, but the structure of the algorithm is the same on both sides. 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 [13], 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.

It must be noted that the checksum and each summary tables are built and then queried once: thus computing an index to help access some data is not beneficial as its building cost would not be amortized.

VI. IMPLEMENTATIONS

We know of three implementations of our algorithm. We developed a proof of concept free software prototype [18] targeting both PostgreSQL and MySQL, including actually synchronizing tables between both systems. Vandiver [10] developed a version in Java in order to compare his algorithm with ours in his PhD thesis, but the corresponding code is not available. Finally, Nacos [19] built a C-coded trigger-based tool using one big summary table, while the reconciliation algorithm is also in Java.

When considering an implementation of our algorithm, several key functionalities are needed: 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: SQL `COALESCE` function can be used to substitute NULL values. Second, we use a shortened version of the `MD5` cryptographic hash function, which is available on both systems as a checksum. However, in order to have results comparable between PostgreSQL and MySQL, a lot of hopus-pocus was necessary for casting, converting and truncating the results reliably. We added cast functions to PostgreSQL and developed special conversion functions for MySQL. As the comparison may be CPU bound, especially on a local network, we also provide a fast although not cryptographically secure checksum function as extensions to both database systems. Third, a criterion must be chosen to group the tuples in the tree building phase, compatible with the data type holding the key checksum. Our implementation store results as 8-byte integers, and we restricts block sizes to power of 2, so that we can use simple mask operations. Finally, a checksum combining aggregate function must be provided. Our implementation uses either SUM or XOR.

The table comparison can be restricted to perform partial checks: The comparison to be performed is specified through URL-formed command line arguments with reasonable defaults, and an option can select a subset of tuples with a `WHERE` clause.

Another implementation issue is whether to use threads to parallelize the queries on both sides, especially the initial checksum table building which represent the largest computation cost. This can influence significantly the performance up to a factor of two on a fast network. However, on a slow network, most of the time is spent in communications with a bandwidth shared by the two parallel connections, and the impact on performance is small or null. Our perl script

```
SELECT COALESCE(T1.key, T2.key) AS key,
  CASE WHEN T1.key IS NULL THEN 'DELETE'
       WHEN T2.key IS NULL THEN 'INSERT'
       ELSE 'UPDATE'
  END AS operation
FROM T1 FULL JOIN T2 ON ((T1.key)=(T2.key))
WHERE T1.key IS NULL       -- DELETE
   OR T2.key IS NULL       -- INSERT
   OR (T1.val) <> (T2.val) -- UPDATE
```

Fig. 7. Local comparison of tables `T1` and `T2` (with `NOT NULL` attributes)

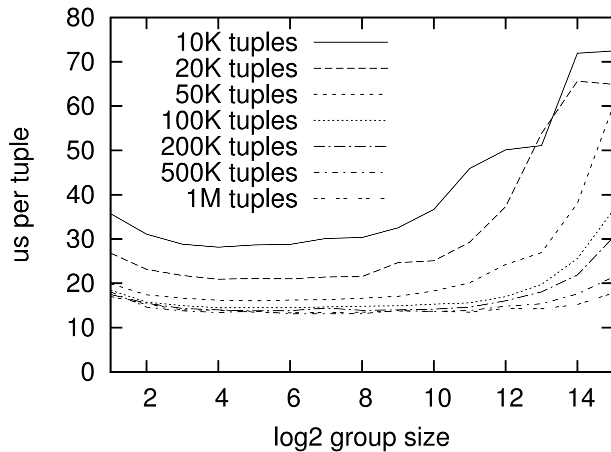

Fig. 8. Comparison time per tuple, 1 Gb/s, 3 diffs

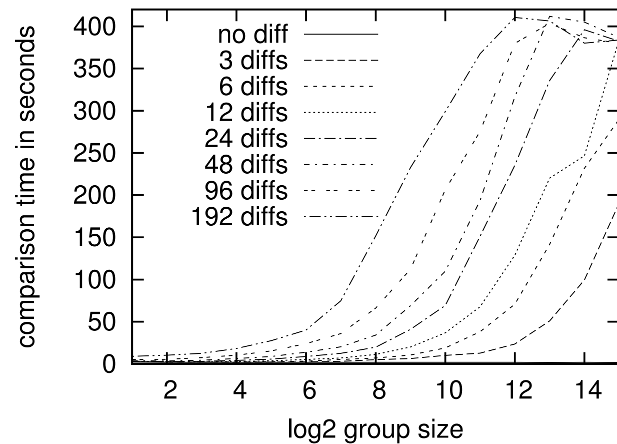

Fig. 10. Comparison time vs block size, 100 K tuples, 100 Kb/s bandwidth

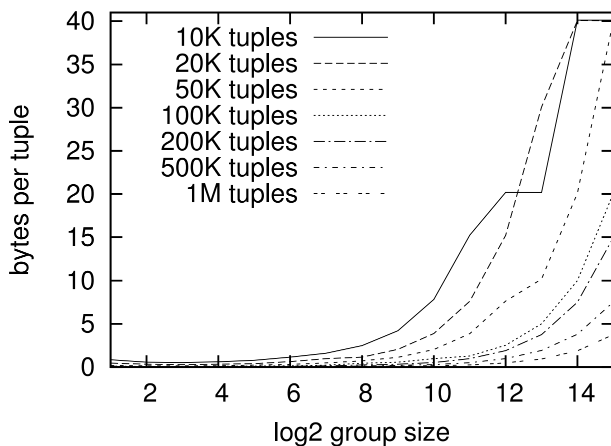

Fig. 9. Volume transferred per tuple, 1 Gb/s, 3 diffs

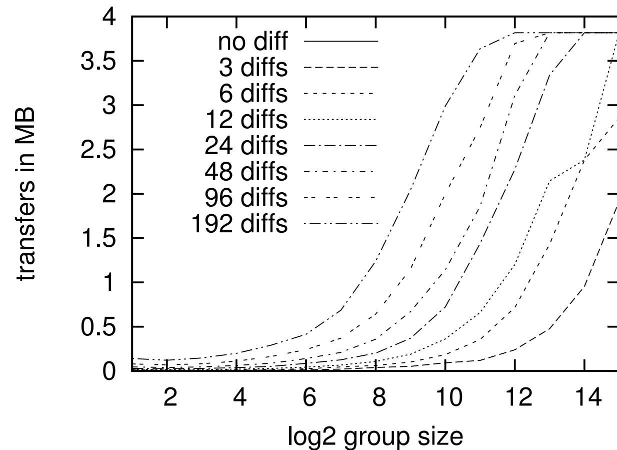

Fig. 11. Volume transferred vs block size, 100 K tuples

implementation includes a threading option, but it does not work with the PostgreSQL because of a bug in the standard driver implementation.

VII. EXPERIMENTS

We report in the following about 5500 runs of our comparison algorithm performed with our implementation on PostgreSQL databases using the fast checksum, on randomly generated tables from 10 K top 1 M rows of 450-byte long records, compared with varying block sizes, and in a bandwidth bound environment. Three Linux desktop computers in an isolated network were used, two of them holding the databases, the third performing the reconciliation. The network bandwidth was controlled on every connection with the `tc` traffic control command. The base case for a remote comparison algorithm would be to transfer all tuples on the other side, and then perform the comparison locally with an external join as suggested in Figure 7.

The following measures must be taken with caution: they are sensitive to many parameters such as hardware including hard disk, cpu and memory performance, system disk cache management, database configuration and optimization, algorithm implementation details such as threading, network latency, bandwidth, mtu and congestion status, as well as various protocol overheads. To encompass all these, we used overall elapsed times: it covers both computation and network, the preeminence of which varies depending on the actual test conditions.

Figures 8 and 9 show normalized data collected from a 1 Gb/s local area network. The horizontal axis is the $\log_2$ group size used for building the summary tree. The vertical axis in the normalized time to perform the reconciliation in μs per tuple for the first figure, and the number of bytes per tuple (without overheads) transferred for the second. Different table sizes are investigated to recover 3 differences. The two small table sizes incur high latency and summary computation overheads. For other larger sizes, the comparison requires a somehow constant $14\,\mu$s/tuple. As there are few differences, the main costs are in computing the checksum and summary tables and in network latency, especially for small group sizes. The drilling down is mostly negligible but for large group sizes where big chunks of checksums are fetched. The base case of transferring data alone is about $4.5\,\mu$s/tuple in this setting, thanks to the high bandwidth.

In Figures 10 and 11, a 100 Kb/s network link is used to reconcile 100 K tuple tables with different block parameters.

Copyright (c) IARIA, 2011 ISBN:978-1-61208-115-1

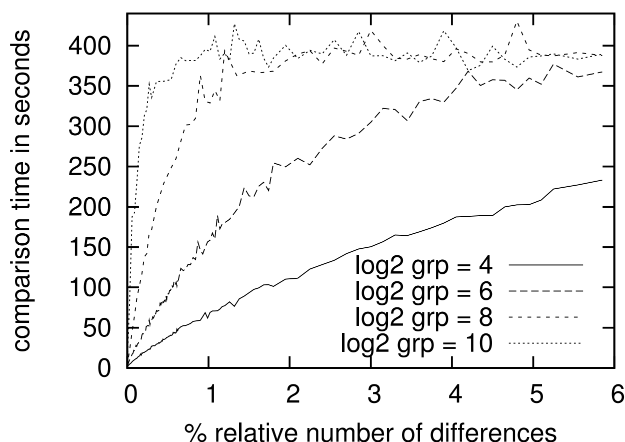

Fig. 12. Comparison time vs diffs, 100 K tuples, 100 Kb/s bandwidth

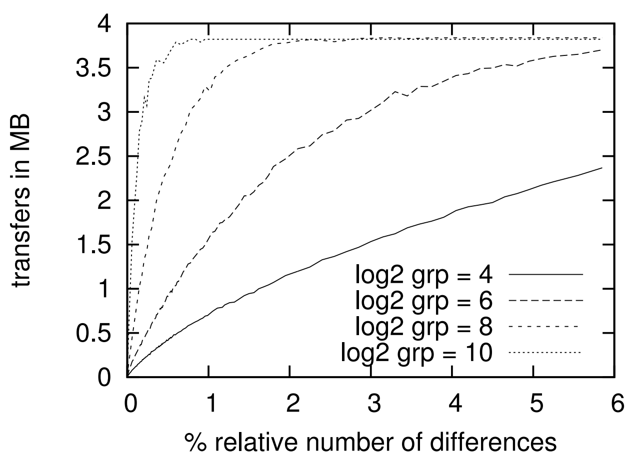

Fig. 13. Volume transferred vs diffs, 100 K tuples

The transfer of all data would require about 4000 seconds, so the 2 seconds comparison time achieved is a 2000-fold speedup. The number of differing tuples is made to vary from 0 to 192, and the comparison time is displayed in seconds. Block sizes from $8 = 2^3$ to $126 = 2^7$ perform best. Larger block values are loaded by the requirement in network bandwidth and as the number of differences increases. There is a saturation when all checksums are fetched from T_0.

A 100 Kb/s bandwidth network is also used in Figures 12 and 13. The number of differences is scaled up for various block sizes and 100 K tuples, and shown as a percentage relative to the table size. The linear dependency of the algorithm in a network bound context shows up for small number of differences, and for higher figures the saturation effect is encountered when all checksum blocks are fetched. The turn around for saturation is expected when all checksums are fetched, that is at about size $n/2^f$, *i.e.*, 6%, 1.5%, 0.4% and 0.1% for the four block sizes presented. The base case in this setting is about 1 hour to transfer the table, while 7 minutes suffice to identify the differences based on checksums.

Vandiver [10] also performed experiments with his implementation of our algorithm, and compared them to his own: The main costs comes from computing the checksum table. The drilling through the data structures requires few communications. When considering highly correlated faults, our generic approach can be beaten because the key randomization in groups becomes a liability.

VIII. CONCLUSION

We have presented an algorithm dedicated to remote relational database table comparisons with a parametric block size. Keys of inserted, deleted or updated tuples are identified quickly. 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 performs a reconciliation merge of partial checksums fetched level by level. All experiments of our remote comparison algorithm show better performances than the brute force download solution, but for the Gb/s local network. This shows that our implementation provides a generic, elegant and portable solution to remote comparison of database tables.

Acknowledgment – Thanks to Laurent Yeh for pointing out relevant related work, and to Michael Nacos and Benjamin M. Vandiver for local or remote discussions.

REFERENCES

[1] T. Suel and N. Memon, *Lossless Compression Handbook*. Academic Press, 2002, ch. Algorithms for Delta Compression and Remote File Synchronization.
[2] V. Chvatal, D. A. Klarner, and D. E. Knuth, "Selected combinatorial research problems." Stanford University, Tech. Rep., 1972.
[3] J. J. Metzner, "A parity structure for large remotely located replicated data files." *IEEE Trans. Computers*, vol. 32, no. 8, pp. 727–730, 1983.
[4] A. Tridgell and P. MacKerras, "The rsync algorithm," Australian National University, TR-CS 96-05, Jun. 1996.
[5] A. Tridgell, "Efficient algorithms for sorting and synchronization," Ph.D. dissertation, Australian National University, 1999.
[6] K. A. S. Abdel-Ghaffar and A. E. Abbadi, "An optimal strategy for comparing file copies," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 1, pp. 87–93, 1994.
[7] M. Karpovsky, L. Levitin, and A. Trachtenberg, "Data verification and reconciliation with generalized error-control codes," *IEEE Transactions on Information Theory*, vol. 49, no. 7, pp. 1788–1793, Jul. 2003.
[8] Y. Minsky, A. Trachtenberg, and R. Zippel, "Set reconciliation with nearly optimal communication complexity," *IEEE Transactions on Information Theory*, vol. 49, no. 9, pp. 2213–2218, Sep. 2003.
[9] A. Trachtenberg and Y. Minsky, "Efficient Reconciliation of Unordered Sets," Cornell University, Tech. Rep. 1778, Nov. 1999.
[10] B. M. Vandiver, "Detecting and Tolerating Byzantine Faults in Database Systems," Ph.D. dissertation, MIT, Cambridge, MA, USA, Jun. 2008, MIT-CSAIL-TR-2008-040.
[11] J. W. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed content delivery across adaptive overlay networks," *IEEE/ACM Transactions on Networking*, vol. 12, no. 5, pp. 767–780, Oct. 2004.
[12] R. C. Merkle, "Secrecy, authentication and public key systems / a certified digital signature," Ph.D. dissertation, Stanford University, 1979.
[13] G. Maxia, "Taming the distributed database problem: A case study using MySQL," *Sys Admin*, vol. 13, no. 8, pp. 29–40, Aug. 2004.
[14] B. Schwartz, "MySQL toolkit: `mk-table-sync`," http://www.maatkit.org/, Mar. 2007, checked 2010-11-11.
[15] ——, "An algorithm to find and resolve data differences between mysql tables," Blog on http://www.xaprb.com, Mar. 2007, checked 2010-11-11.
[16] DKG Advanced Solutions, "DBDiff for windows," http://www.dkgas.com/, 2004, checked 2010-11-11.
[17] EMS, "Db comparer," http://www.sqlmanager.net/, 2006, checked 2010-11-11.
[18] F. Coelho, "PgComparator," Software available on http://pgfoundry.org/projects/pg-comparator, Aug. 2004, version 1.7.0 on 2010-11-12.
[19] M. Nacos, "Pg51g," http://pgdba.net/pg51g/, Sep. 2009, checked 2010-11-11.

Copyright (c) IARIA, 2011 ISBN:978-1-61208-115-1