Skew Handling Techniques in Sort-Merge Join

Wei Li and Dengfeng Gao and Richard T. Snodgrass

TR-62

A TIMECENTER Technical Report

Title **Skew Handling Techniques in Sort-Merge Join**

Author(s) Wei Li and Dengfeng Gao and Richard T. Snodgrass

Publication History June 2001. A sc TimeCenter Technical Report.

TIMECENTER Participants

Aalborg University, Denmark
Christian S. Jensen (codirector), Michael H. Böhlen, Heidi Gregersen, Dieter Pfoser, Simonas Šaltenis, Janne Skyt, Giedrius Slivinskas, Kristian Torp

University of Arizona, USA
Richard T. Snodgrass (codirector), Dengfeng Gao, Vijay Khatri, Bongki Moon, Sudha Ram

Individual participants
Curtis E. Dyreson, Washington State University, USA
Fabio Grandi, University of Bologna, Italy
Nick Kline, Microsoft, USA
Gerhard Knolmayer, Universty of Bern, Switzerland
Thomas Myrach, Universty of Bern, Switzerland
Kwang W. Nam, Chungbuk National University, Korea
Mario A. Nascimento, University of Alberta, Canada
John F. Roddick, University of South Australia, Australia
Keun H. Ryu, Chungbuk National University, Korea
Michael D. Soo, amazon.com, USA
Andreas Steiner, TimeConsult, Switzerland
Vassilis Tsotras, University of California, Riverside, USA
Jef Wijsen, University of Mons-Hainaut, Belgium
Carlo Zaniolo, University of California, Los Angeles, USA

For additional information, see The TIMECENTER Homepage:
URL: <http://www.cs.auc.dk/TimeCenter>

The TIMECENTER icon on the cover combines two "arrows." These "arrows" are letters in the so-called *Rune* alphabet used one millennium ago by the Vikings, as well as by their precedessors and successors. The Rune alphabet (second phase) has 16 letters, all of which have angular shapes and lack horizontal lines because the primary storage medium was wood. Runes may also be found on jewelry, tools, and weapons and were perceived by many as having magic, hidden powers.

The two Rune arrows in the icon denote "T" and "C," respectively.

Abstract

Joins are among the most frequently executed operations. Several fast join algorithms have been developed and extensively studied; these can be categorized as sort-merge, hash-based, and index-based algorithms. While all three types of algorithms exhibit excellent performance over most data, ameliorating the performance degradation in the presence of skew has been investigated only for hash-based algorithms. This paper examines the negative ramifications of skew in sort-merge, and proposes several refinements of sort-merge join that deal effectively with data skew. Experiments show that some of these algorithms also impose virtually no penalty in the absence of data skew, and are thereby suitable for replacing existing sort-merge implementations in relational DBMSs. We also show how band sort-merge join performance is significantly enhanced with these refinements.

1 Introduction

Because joins are so frequently used in relational queries and because joins are so expensive, much effort has gone into developing efficient join algorithms. The simple nested-loop join is applicable in all cases, but imposes quadratic performance. For equijoins, sort-merge join was found to be much more effective, with excellent performance over a wide range of relation sizes, given adequate main memory. Later, researchers became interested in hash-based join algorithms and it has been shown that in many situations, hash-based algorithms perform better than sort-based algorithms. However, there exist cases in which the performance of hash-based joins falls short. If there are several relations that will participate in multiple joins, the "interesting order" will often determine that sort-based join is better, to enable the joins to run in a pipeline fashion [Selinger et al. 79], because the output of sort-merge join is sorted, thereby possibly obviating the need for sorting in subsequent sort-merge joins. Graefe has exposed many dualities between the two types of algorithms and their costs differ mostly by percentages [Graefe 94, Graefe et al. 94]. Most DBMSs now include both sort- and hash-based, as well as nested-loop and index-based join algorithms.

The distribution of the input data values can have a dramatic impact on the performance of both sort- and hash-based algorithms. The term "skew" involves several related but different effects. The most fundamental distinction is that between partition skew and intrinsic skew [Walton et al. 91].

Partition skew is of concern in hash-based join. In the first step of hash join, tuples are hashed into the corresponding bucket that is computed by the hash function. However, an attribute being hashed may not be uniformly distributed within the relation, and some buckets may then contain more tuples than other buckets. When this disparity becomes large, the bucket no longer fits in main memory, and hash-based join degrades into nested-loop join. Partition skew originates in the hash function chosen by the optimizer; there may exist other hash functions that better randomize the input. In parallel systems, partition skew may result in an unbalanced workload, which can greatly degrade the performance of the whole system. Several papers have proposed ways to deal with partition skew in hash-based join [DeWitt et al. 92, Hua and Lee 91, Kitsuregawa et al. 89, Nakayama et al. 88, Walton et al. 91].

Intrinsic skew occurs when attributes are not distributed uniformly; it has also been called *attribute value skew* [Walton et al. 91]. Intrinsic skew impacts the performance of both hash and sort-based joins. Sort-merge join works best when the join attributes are the primary key of both tables. This ensures that there are no duplicates present, so that a tuple in the left-hand relation will join with at most one tuple in the right-hand relation, avoiding intrinsic skew. When an equi-join is performed over non-key attributes, intrinsic skew is generally present. Inequality predicates, such as found in *band join* (to be discussed in detail later), in *temporal join* [Soo et al. 94] and *temporal Cartesian product* [Zurek 96], and in *multi-predicate merge join* proposed for containment queries on XML data [Zhang et al. 01], exacerbate the problem.

The general advice is to use sort-merge join in the presence of significant intrinsic skew, because bucket overflow is so expensive. However, we are aware of no papers on either the impact of intrinsic skew on the

performance of (centralized) sort-merge join, nor on ways to deal with such skew. In fact, the classical sort-merge algorithm presented in many database textbooks yields incorrect results in the presence of intrinsic skew.

In this paper we provide a variety of algorithms that correctly contend with intrinsic skew in sort-merge join. For the remainder, the term "skew" will denote intrinsic skew, and "join" will refer to sort-merge join (also called merge-join or sort-join, in several variants). Section 2 identifies the three problems that skew presents to sort-merge join, and shows how two of these problems can be solved. Section 3 is the core of this paper, proposing eight variants of sort-merge join, all operating correctly in the presence of all three types of skew. The following section compares the performance of these algorithms. Section 5 shows how the algorithms perform for band joins [DeWitt 91], in which substantial skew is invariably present. Finally, Section 6 concludes with a recommended replacement for the traditional sort-merge join algorithm.

It may be surprising that anything new can be said about the venerable sort-merge join. The problem with intrinsic skew in sort-merge join is almost certainly known by vendors, though there is little in the extant literature about this problem. Our discussions with vendors indicate that some DBMSs fall back to nested loop when problematic skew is encountered, or shift tuples back in memory so that more tuples can be read, which allows greater skew to be accommodated, but doesn't solve the full problem (we examine shifting tuples in Section 3.3). Another contact told us that commercial "sort-merge algorithms at least use some variant of the R-1 approach" we introduce below. However, as we show, R-1 and a multi-run variant R-n are not competitive in performance when compared to the other algorithms we propose. In any case, the present paper is the first to carefully examine precisely when skew becomes a problem in sort-merge join, the first to present specific algorithms to address these problems, and the first to analyze the performance implications of these approaches. We feel that our recommended replacement could ensure a more accurate analysis of inequality join algorithms, and could enhance the performance of commercial sort-merge join implementations.

2 Preliminaries

The join algebraic operator takes two input relations, of arity m and n, and produces a single resulting relation. A wide variety of joins have been defined, including equijoins, natural joins, semi-joins, outer joins, and composition [Mishra and Eich 92]. We will consider the general case in which the join outputs all the attributes (hence, the arity of the result is $m + n$). We assume that the join has an explicit predicate containing at least one equality test between attributes of the two underlying relations (these attributes are termed *equijoin attributes*: EA); sort-merge join is applicable only in the presence of such equijoin attributes. We term the (optional) remainder of the join predicate the *supplemental predicate* (SP), which can involve equality comparisons between attributes of one of the input relations, either with themselves or with constants, as well as inequality comparisons and function invocations. The supplemental predicate can significantly reduce the size of the resulting relation, especially if the equijoin attributes do not constitute a primary key of either of the underlying relations.

The traditional sort-merge algorithm is usually shown as in Figure 1(a), in which pL is a pointer into relation L, and similarly with pR, each ranging from 1 to the cardinality of the relation; $L[pL]$ is the tuple at position pL; and $L[pL](EA)$ are the value(s) of the equijoin attribute(s) of that tuple. In this algorithm, the sequence of attributes on which the sort is applied is not important. (From now on, we assume a single equi-join attribute.) Sort-merge join preserves the sort order of the inputs, a useful property to exploit in the presence of multiple joins.

In this context, *skew* is the presence of multiple tuples in L or R with identical values for the equijoin attribute. These tuples, collectively called a *value packet* [Graefe 93, Kooi 80] for each such value, are contiguous in the input relations after they are sorted. So an equivalent definition of skew is the presence

Traditional Sort-Merge Join :

Sort relation L on the attribute EA
Sort relation R on the attribute EA
$pL \leftarrow 1$
$pR \leftarrow 1$
repeat until $pL = L$.length **or** $pR = R$.length
  if $L[pL](EA) = R[pR](EA)$
    if $SP(L[pL], R[pR])$ output($L[pL] \circ R[pR]$)
    advance pR
  else if $L[pL](EA) > R[pR](EA)$
    advance pR
  else // $L[pL](EA) < R[pR](EA)$
    advance pL

Advance pL *:*

$pL \leftarrow pL + 1$

(a)

Traditional Sort-Merge Join With Skew:

Sort relation L on the attribute EA
Sort relation R on the attribute EA
$pL \leftarrow 1$
$pR \leftarrow 1$
repeat until $pL = L$.length **or** $pR = R$.length
  if $L[pL](EA) = R[pR](EA)$
    $pR2 \leftarrow pR$
    repeat
      if $SP(L[pL], R[pR2])$ output($L[pL] \circ R[pR2]$)
      advance $pR2$
    until $L[pL](EA) \neq R[pR2](EA)$
    advance pL
  else if $L[pL](EA) > R[pR](EA)$
    advance pR
  else // $L[pL](EA) < R[pR](EA)$
    advance pL

(b)

Figure 1: Traditional sort-merge join Algorithms, original (a) and accommodating skew (b)

Advance pL *:*

$pL \leftarrow pL + 1$
if $pL = B + 1$
  read next block of L into BL
  $pL \leftarrow 1$

Figure 2: Rendering sort-merge join block-based

of a value packet containing more than one tuple. The traditional algorithm must be modified to backtrack, yielding the algorithm in Figure 1(b). This algorithm effectively applies nested loop (cf. the nested repeats) on value packets it encounters, applying the supplemental predicate, if present, to each pair of tuples, one from each value packet. pR records where the value packet starts in R; $pR2$ iterates over the value packet.

This algorithm as presented in Figure 1(b) is *tuple-oriented*: the input relations are treated as in-memory arrays of tuples. Join implementations are always *block-based*, in which a block of tuples is read into main memory, to be processed and then replaced with successive blocks read from disk. The algorithm in the figure can be rendered block-based by simply inserting block reads (to an in-memory array, either BL or BR, of size B) whenever a pointer is indexed out of the in-memory block, as shown with new code for *advance* in Figure 2 and changing references of L to BL and of R to BR.

This is where most presentations of sort-merge move on to a complexity analysis of the algorithm. Unfortunately, making this straightforward change breaks the algorithm when skew is present. There are three types of skew:

1. skew occurring only in the left-hand side (LHS) relation,
2. skew occurring only in the RHS relation, or
3. skew occurring in both the LHS and RHS relations.

The problem arises when a value packet crosses a block boundary. (A value packet entirely contained in a block presents no problem.) These three cases are shown in Figure 3. In this figure, each rectangle denotes a buffer's worth of tuples. **A** denotes a value packet with an equijoin attribute value of A; similarly, **B** denotes a value packet with an equijoin attribute value of B. The arrows in the figure denoting the reading pointer (pL or pR) into the block.

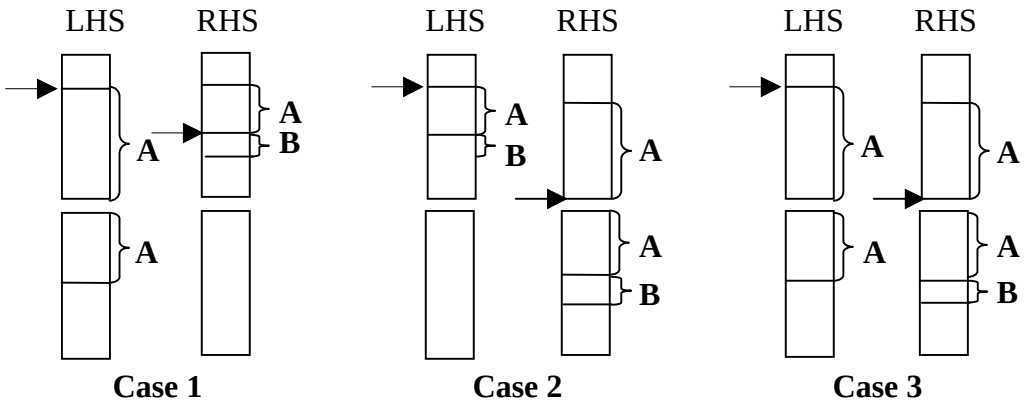


Figure 3: Types of skew in sort-merge join

Graefe mentioned the skew problem and indicated that one of two merging scans must be backed up when both inputs contain duplicates of a join attribute value and when the specific one-to-one match operation requires that all matches be found, not just one match [Graefe 93]. Mishra and Eich also address this problem: if the join attributes are not the primary key attributes, several tuples with the same attribute value may exist [Mishra and Eich 92]. This necessitates several passes over the same set of tuples of the inner relation. So whenever they encounter a duplicate LHS value, they state that it is necessary to backtrack to the previous starting point in the RHS relation, but don't provide any details. The only algorithm for

handling skew in a block-oriented environment that we have found is in Garcia-Molina's book, which we will consider further in Section 3.4.

We now consider in detail how to contend with these three sources of skew. The first case of skew, which we term *LHS skew*, presents no problem, as the block boundary is encountered in the outer loop. The next block of the LHS is read in, and the join continues with the same value packet in the RHS. Skew in the RHS (either alone, termed *RHS skew*, or in conjunction with LHS skew, termed *dual skew*) does cause problems, because the block boundary is encountered within the inner loop. In the presence of RHS skew, the next block of the RHS is read in while the right-hand value packet is being joined with the first tuple of the left-hand value packet, which renders the previous block inaccessible for joining with the remaining tuples in the value packet in the LHS. The right-hand pointer can only be moved back to the start of the block, so subsequent tuples in the left-hand value packet will only be joined with the second portion of the right-hand value packet.

This implies that if skew is known to be absent from one of the underlying relations, for example if the equijoin attributes form a primary key of a relation, then that relation should be placed as the RHS of the skew, which is always possible because of the commutativity of join, though that may have implications on the efficiency both of the join in question and other joins in the query. Doing so, however, doesn't solve the problem in the general case; we still need a join method that can accommodate skew, especially as often the reason sort-merge join is considered in the first place is that the skew argued against adopting hash join.

There is one additional complication that will become relevant. Recall that sort-merge join uses a disk-based sorting phase that starts by generating many small fully-sorted *runs*, merging these into longer runs until a single run is obtained (this is done for the left-hand side and right-hand side independently). Each step of the sort phase reads and writes the entire relation. The merge phase then scans the totally-sorted left and right-hand relations to produce the output relation.

A common optimization is to stop the sorting phase one step early, when there are a small number of fully sorted runs. The final step is done in parallel with the merge phase of the join, thereby avoiding one read and one write scan. This optimization impacts how dual skew is accommodated.

3 New Sort-Merge Join Algorithms that Deal With Skewed Data

The goals of the new algorithms are to incur no disk overhead under low skew and perform efficiently under heavy skew.

3.1 Reread with One Run (R-1) and with Multiple Runs (R-*n*)

R-1 is a simple extension of Figure 1(b)+Figure 2, in which blocks of the RHS are reread whenever a reference is made to a tuple in a block that was previously replaced with a subsequent block (during the advancing of $pR2$). This algorithm exploits the presence of only two runs to be merged, one each from the LHS and the RHS, which means that there will be a large buffer for R, reducing the need for rereading. The down side is that an extra pass is needed to produce a single run for the right hand side only.

To accommodate RHS and dual skew, the R-1 algorithm reads blocks when necessary (when a pointer is incremented past the end of a block), and rereads blocks of the RHS when pR is reset to the beginning of a value packet, an event termed a *hiccup*. The hiccup comes in the middle of the algorithm, when the disk block of RHS that started the value packet is reread.

R-1 can be considered to be the minimal extension of the standard sort-merge join that correctly deals with all three kinds of intrinsic skew. As we'll see later in the paper, the performance of this algorithm degrades very quickly in the presence of skew.

R-n is a variant that supports multiple runs on the right-hand side, with rereading on a per-run basis. (The "n" simply means "multiple runs'," as contrasted with a single run on each side.) With multiple runs, the rereading process becomes more complex. For each RHS run, we record the backup pointer ($pR2$ in Figure 1(b)). Every time we increment the LHS pointer, we check the recorded information to see whether it is necessary for each RHS to reread the block containing the initial tuple of the value-packet (the runs for which the value packet does not entirely fit in the run's block will have to be reread). If there is no skew, we still need to check this information, which represents CPU overhead. Skew will probably generate more random reads, since the skewed data is likely to be spread across several runs.

3.2 Block-based Reread with One Run (BR-1) and with Multiple Runs (BR-*n*)

While R-1 and R-n were block-based in terms of their non-skew portion, they are both tuple-based in terms of their rereading: a hiccup occurs for the second and successive tuples in the LHS value packet. We now present two further refinements (BR-1 and BR-n) that are entirely block-based.

The discussion in Section 2 differentiated RHS and dual skew. To address the simpler of the two, RHS skew, we break the nested loop into two parts, joining the left value packet with the portion of the right value packet in the buffer before moving on to the next right-hand buffer. To distinguish between cases 2 and 3 (RHS skew and dual skew), we adopt a *prediction rule*: dual skew is present if the value of the last tuple in the left block matches (for the equijoin attribute) that of the value of the last tuple of the right block.

BR-n avoids the last run merge by storing information about the state of each LHS run. We divide each LHS run into three parts according to the join situation in the run's buffer. The partition is shown in Figure 4. Each buffer is divided into three areas: *finished*, *pending*, and *next-start*. Within the finished area all tuples have completed all their joins. The pending area contains the tuples that have been joined with some of the RHS tuples. Within the next-start area, none of the tuples have been joined with RHS tuples.

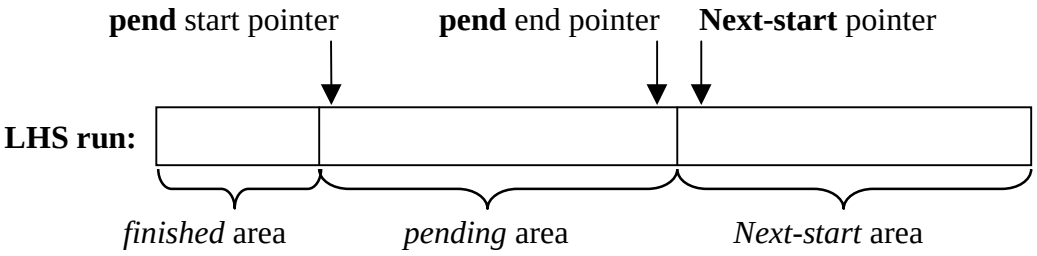


Figure 4: Divide each LHS buffer into 3 parts

In this algorithm, shown in Figure 5, the LHS and RHS in-memory blocks are denoted by BL and BR, respectively, with a block size of B. pL, pR and $pR2$ range from 1 to B, pointing into the main-memory block of L or R. $pR2$ points to the first tuple in the main-memory portion of the value packet for the RHS; pL and pR range over the value packets of LHS and RHS. The innermost nested loops ensure that the left value packet is joined with the portion of the right value packet in the buffer before moving on to the next right-hand buffer.

The status of each LHS run will be one of the following three possibilities.

- **non-reread**—the whole buffer has not been joined
- **full-reread**—the buffer has both pending and next-start information
- **next-start**—the buffer only contains next-start information

Block-based Sort-Merge Join :

```
Sort relation L on the attribute EA
Sort relation R on the attribute EA
read first block of L into BL
pL ← 1
read first block of R into BR
pR ← 1
repeat until finished with L
    pR2 ← ⊥
    repeat until finished with R
        if BL[pL](EA) < BR[pR](EA)
            break
        else if BL[pL](EA) > BR[pR](EA)
            advance pR
        else                                              // match
            if pR2 = ⊥
                pR2 ← pR                                  // remember start of value packet
            if SP(BL[pL], BR[pR]) output(BL[pL] ∘ BR[pR])
            if pR ≠ B                                     // end of RHS block
                advance pR
            else
                pred ← (BL[B](EA) = BR[B](EA))
                repeat until no matching tuple exists in RHS   // finish off value packet
                    join all the matching tuples in BL and BR
                    read the next block of RHS
                if pred                                   // dual skew
                    advance pL                            // to next block of LHS
                    pR ← pR2                              // restore RHS pointer
                else                                      // RHS skew only
                    advance pL                            // to new value packet
                    advance pR                            // to new value packet
                    pR2 ← ⊥
    if pR2 ≠ ⊥
        pR ← pR2
    advance pL
```

Figure 5: Block-based reread with one run, BR-1

With this augmented status, we can make the following revisions on the R-n algorithm to realize the BR-n algorithm.

- Each time we start with another tuple from the LHS to join with the RHS runs, we adjust the reading pointer for the RHS runs according to the reread information in the LHS runs. If there is no reread information, we would still use the same reread method as in BR-1.
- When any RHS run reaches the end of a block, for this run, do the following for each of the LHS runs, according to their status:
  - **non-reread**: join the run from the current reading pointer with the RHS run and record the finishing pointer. According to the two pointers, change the status to pending and set up the pending and next-start reread information.
  - **full-reread**: join the tuples in the pending area with the RHS run and move forward the reread pointer for the RHS run.
  - **next-start**: join the tuples in the next-start area with the RHS run and change the run's status to full-reread; then set the corresponding pending information according to the join results and the next-start information.
- When any LHS run reaches the end of a block, check whether there is next-start information in that run. If so, forward the next-start information to the next block.

A note on the implementation: the prediction rule requires that we look at the last record in the block. This is easy for fixed-length records, but more difficult for variable length records. To avoid the CPU overhead needed to search from the first record to the last record in the block, the offset of the last record can be recorded in each block when writing out the run in the sort phase.

BR-n handles hiccups on a block-by-block basis, across many runs. Even with this optimization, hiccups are still quite expensive. The next four algorithms attempt to avoid hiccups in the presence of skew.

3.3 Block-based Reread with Smart Use of Memory on Multiple Runs (BR-S-*n*)

Although BR-n avoids hiccups in the presence of RHS skew, it has to reread when it encounters dual skew. To address dual skew with less rereading, we can make better use of the main memory buffer.

As in Figure 6, when the end of the RHS block is hit and dual skew is detected, we clearly know all the tuples preceding the current value packet have been joined and need not be kept in the memory. Thus, these tuples can be discarded and their space can be used to hold the tuples in the current value packet that resides in the successive block. This involves shifting the current value packet, which resides at the very bottom of the run in main memory, to the top of the buffer, then reading in more of the value packet into the free area below. From our previous analysis, the buffer should be relatively large; in most cases, we can accommodate all the skewed data in one block of buffer and thus avoid rereading altogether.

In the extreme situation that the size of skewed data exceeds the block size, the reread can't be avoided but we are careful to start the read in the block in which the value packet starts, so that the tuples preceding the most recent value packet are not reread: only the most recent value packet and subsequent blocks need to be reread.

3.4 Spooled Cache for Skewed Data with One More Pass on LHS (SC-1)

In their book, Garcia-Molina et al. [93] recommend in the case of skewed input that main-memory use for other aspects of the algorithm be reduced, thus making available a potentially large number of blocks to

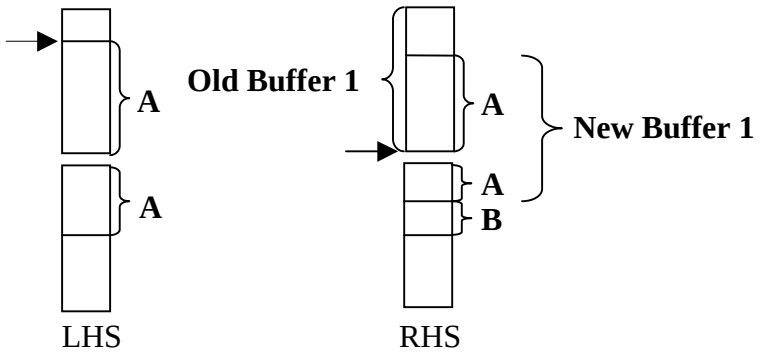


Figure 6: Smart Use of Block Space

hold the tuples in a given value packet. Subsequent blocks of both the LHS and RHS value packets are read in, replacing blocks already scanned. If the skew data still do not fit in main memory, nested loop has to be used.

As they note, this algorithm is difficult to generalize to multiple runs. Also, the details are not presented, nor is a performance study. Here we present a related approach that we extend in the next section to support multiple runs.

The basic idea is to add another buffer in main memory, termed the *join-condition cache*, to hold the skewed tuples. Specifically, the join-condition cache holds tuples from the RHS that satisfy the join condition and have not been completely joined matched yet with tuples from the corresponding value packet in the LHS. The size of the cache can be specified before the join, or it can be expanded incrementally in the join. However, there always exists the possibility that the cache may overflow. At the cache's overflow point, we have to make a decision: either spool the cache data to the disk or use rereading to present the cache from overflowing. Here, we adopt the first approach; in Section 3.6, we will adapt the algorithm to ensure the cache never overflows, by rereading. We consider the one-run variant here; the next section generalizes this spooled cache approach to multiple runs.

We need to handle both RHS and dual skew with the cache (LHS skew is trivially handled). We adapt the prediction rule to introduce another rule which will be helpful for block-based execution.

1. *Prediction rule* — If we find that the RHS buffer contains the skewed data, then before moving the skewed data into the cache for future join, we check the last tuple in LHS buffer to determine if dual skew is present. If so, we need to store the RHS skewed data into the cache. Otherwise, we avoid this overhead by joining all the RHS skewed data with all the corresponding LHS tuples.

2. *Join before storing rule* — Before we put the RHS skewed tuples into the cache, we always join them with the corresponding LHS tuples. Thus we know exactly which LHS tuples the cached tuples have already been joined with. This rule saves a lot of bookkeeping work, which makes the algorithm's logic easier to understand.

This approach ensures the following invariant.

Invariant: tuples in the cache have been joined with all tuples in previously-read LHS blocks

Using the cache and these rules, we can optimize the algorithm in the following ways.

- When we hit the right block end of RHS, join all the tuples in RHS block with corresponding LHS tuples.

- After the block join listed above, we decide whether we should put the RHS skewed tuples into the cache, using the modified prediction rule. This avoids unnecessary movement of tuples into the cache.
- Each time we read a new LHS block, we should join all the tuples in the cache with the tuples in the just-read LHS block.

The SC-1 algorithm is shown in Figure 7.

3.5 Spooled Cache for Skewed Data on Multiple Runs: SC-*n*

SC-1 assumes that the LHS is only one run, which requires an additional pass to merge the LHS runs. Here we present a revised algorithm, SC-n, which accommodates several LHS runs, while maintaining excellent performance in most situations.

Managing several LHS runs with the prediction rule becomes more difficult, because if any of the LHS runs fail the test, we have to expand the cache. This situation is shown in Figure 8. In the figure, even though run1 and run3 satisfy the prediction test, we still need to put the RHS tuples in the value packet into the cache, because run2 needs to read in new tuples (because the boundary between blocks in run2 inconveniently comes within a value packet), which means that it has just encountered dual skew.

To transition to multiple runs, we must keep track of the state of each LHS run. If there are n LHS runs, we create an array (actually, a bit vector) of size n to record the status of each run, with the following two values.

- **complete**: indicates that the tuples in this run have been joined with *all* the tuples in the cache
- **pending**: indicating that the tuples in this run have *not* been joined with *any* tuples in the cache

Initially, all the runs' status are set to **complete**, since there are no tuples in the cache at the beginning. We wish to ensure the following invariant.

New Invariant:	**complete** runs have been joined with *all* the tuples in the cache, while **pending** runs have been joined with *no* tuples in the cache

With this invariant in mind, we make the following revisions on the algorithm SC-1 to get the new algorithm.

- When any RHS run reaches the end of a block:
  1. Use the prediction rule (is there a run with this value packet in the last tuple?) to check whether we should move the RHS tuples into the cache
  2. If there is no need to expand the cache, then join the RHS tuples with each LHS run
  3. Otherwise,
     - join the RHS tuples for this run with all **complete** LHS runs and
     - move the applicable RHS tuples into the cache
- When any LHS run reaches the end of one block:
  - If this run is a **complete** run, then load the next block for this run and change this run's status to **pending**
  - If this run is a **pending** run, then
    - join all the **pending** LHS runs with the tuples in the cache
    - change all the LHS runs to **complete**

Algorithm SC-1 : *Spooled Join-Condition Cache with One Run*

```
Sort relation L on the attribute EA
Sort relation R on the attribute EA
Merge L so that it has only one run
read first block of L into BL
pL ← 1
read first block of R into BR
pR ← 1
repeat until finished reading L
    pR2 ← ⊥
    repeat until finished reading R
        if BL[pL](EA) < BR[pR](EA)
            break the repeat loop
        elseif BL[pL](EA) > BR[pR](EA)
            advance pR
            continue the repeat loop
        else
            if this is the first matching tuple for BL[pL](EA)
                pR2 ← pR                          // remember start of value packet
            if SP(BL[pL], BR[pR]) output(BL[pL] ∘ BR[pR])
            if pR ≠ B
                pR ← pR + 1
            else                                  // end of R block
                purge the join-condition cache
                join all the corresponding tuples in BL and BR
                if prediction on BL
                    move skewed tuples into cache
                advance pR
                pR2 ← pR                          // modify start position of value packet
    if pR2 ≠ ⊥
        pR ← pR2                                  // restore RHS reading pointer
    advance pL
    if pL = 1                                     // new L block
        join the join-condition cache with all the tuples in L's new buffer
```

Figure 7: Spooled cache algorithm, SC-1

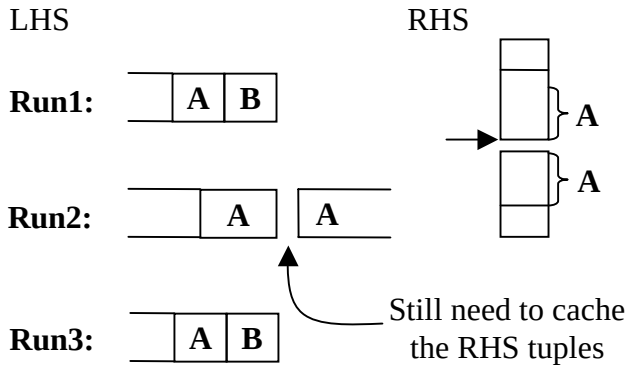


Figure 8: Several runs versus only one run?

- load the next block for the original LHS run (which reached the end), and change its status to **pending**

In this algorithm, purging the cache becomes more complex than in SC-1, in which, we simply purge the cache when we encounter a new RHS value packet. When there are multiple LHS runs, there exists the possibility that there are other runs which may join with tuples being purged from the cache, as shown in Figure 9. This figure shows two value packets, with values A and B. The dotted lines show that tuples from the A value packet from Run1 and Run2 of the LHS have been joined with all the tuples in the cache. The first tuple of the B value packet has just been encountered in Run1 of the LHS. When we reach the end of the second block in the RHS run and we find that the value packet has changed (since it is associated with B values, but the cache contains A values), we purge the cache. But these tuples in the cache have not yet been joined with the value packet(s) in (**pending**) LHS runs. So before purging the cache, we need to join the cache with corresponding tuples in the pending runs, thereby converting them to **complete** runs.

3.6 Block-based Reread with a Non-Spooled Cache on Multiple runs (BR-NC-*n*)

Algorithm R-n (cf. Section 3.1) avoids the overhead of cache maintenance. However, for low skew, this version may cause more disk I/O than algorithm SC-n, which imposes no I/O if the cache can hold all of the skewed data from a value packet. This last algorithm, BR-NC-n, attempts to combine the best features of both the spooled cache and rereading by using a small cache that can deal with low skew in the data distribution. This cache never spools. If cache fills up, we record the cache overflow point and start rereading from that point. Because the cache never reaches the disk, it would not form new "hot points".

4 Evaluation and Comparisons

Among the algorithms we proposed in the previous section, we implemented R-n, BR-n, BR-S-n, SC-1, SC-n and BR-NC-n. We did not implement the simpler R-1 algorithm because preliminary experiments with SC-1 indicated that the additional pass to produce one run of the LHS extracted a high penalty, rendering that algorithm uncompetitive. The results of all the algorithms for the different input relations were compared to ensure that they were identical.

The experiments were developed and executed using the TIME-IT system [Kline & Soo 98], a software package supporting the prototyping of database components. Some parameters are fixed for all the experiments. They are shown in Table 1(a). In all test cases, the generated relations were randomly ordered.

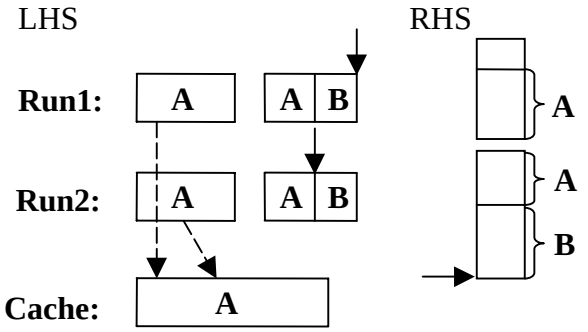


Figure 9: Purging the cache with multiple runs

Parameter	Value
memory size	1MB
cache size	32KB
output buffer size	32KB
page size	1KB
tuple size	128 bytes
join attribute	4 bytes

Metric	Conversion
sequential I/O cost	5 msec
random I/O cost	25 msec
attribute compare	2 μsec
pointer swap	3 μsec
tuple move	4 μsec

Table 1: System characteristics (a) and cost metrics (b)

TIME-IT collects a variety of metrics, shown in Table 1(b); both main memory operations and disk I/O operations were measured. TIME-IT then combines these into a single metric of elapsed time in seconds using the identified weights, thereby not tying the measurements to the underlying processor. We emphasize that this is a computed metric, not actual wall clock time, and so does not capture all of the subtle differences of the algorithms. However, such an approach allows us to understand exactly how each of these metrics is affected by the parameters and by the algorithms.

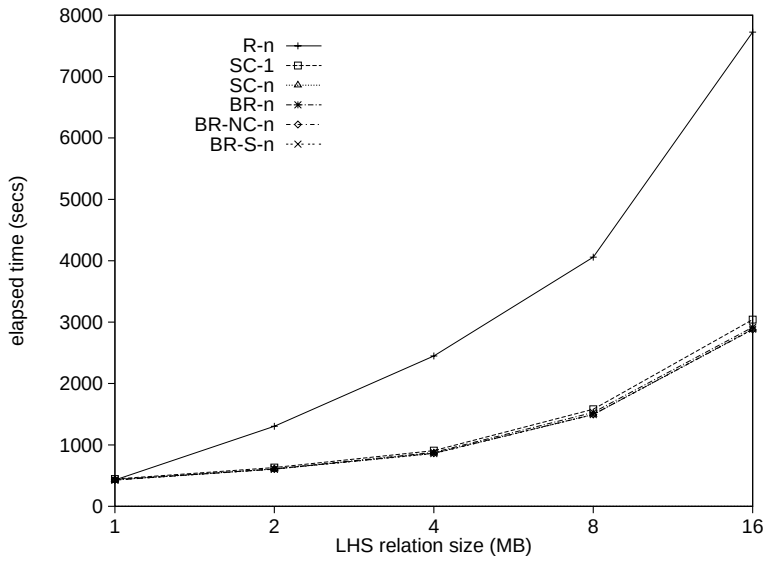
4.1 Experiments

Data skew is the presence of repeated value in the joined attribute. Skew can be realized in a variety of ways. At one end of the spectrum is *smooth skew*, in which some number of tuples have a single duplicate. In smooth skew, some value sets contain two tuples, with the rest containing exactly one tuple. At the other end of the spectrum is *chunky skew* (using a peanut butter metaphor), in which a single attribute value is duplicated many times, thus implying a very large value set. We examine the performance of the various algorithms under these two kinds of skew.

4.1.1 Chunky Skew

In this experiment, we fixed the RHS size at 16MB. The LHS size varies from 1MB to 16MB. The relations on both sides have 1% skew on their join attribute. A relation has 1% chunky skew when only one value of the join attribute repeats and the number of duplicates is 1% of the total number of the tuples in the relation. The results are shown in Figure 10.

R-n behaves terribly when the LHS size increases, because the absolute number of skewed tuples increases when the relation size increases. In fact, it is more than twice as slow as the other algorithms. The number of skewed tuples determines the number of hiccups. We counted the number of hiccups in all the



Algorithm	**Time** (sec)
R-n	7,724
SC-1	3,042
BR-n	2,916
BR-S-n	2,886
SC-n	2,883
BR-NC-n	2,879

Figure 10: Different relation size with chunky skew of 1%

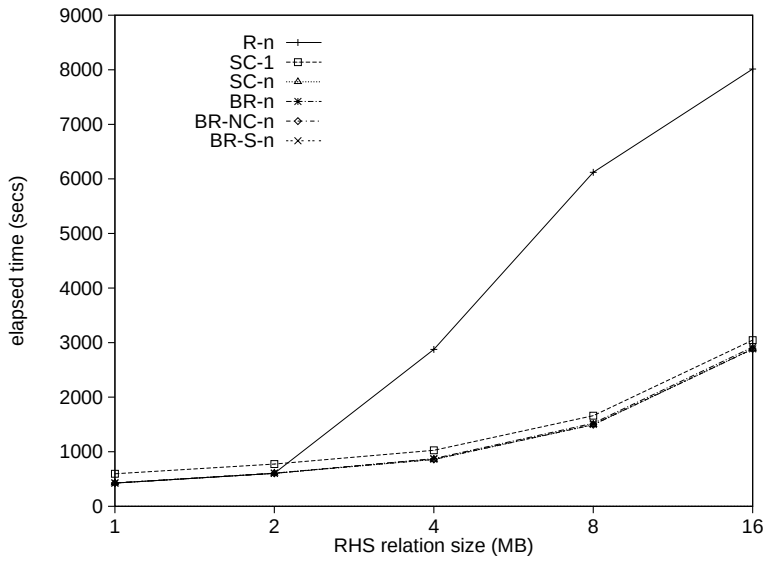
experiments. According to our data, the number of hiccups in R-n surpasses 10000 when the LHS reaches 16MB. For the same relation size, BR-n only exhibits 110 hiccups due to the block-based rereading. The cost of SC-n and BR-NC-n are almost identical to BR-n, because chunky skew can cause cache overflow, which also causes disk operations. From the results, we conclude that the overhead of cache overflow is almost identical to the overhead of block-based rereading. The cost of SC-1 is a little higher (3.5%) because of its one more pass on the left hand side relation. However, this overhead is much less than the tuple-based rereading.

Since SC-1 does one more pass on the left hand side, what will the result be if we put a relatively large relation on the left hand side? We then exchanged the position of left hand side relation and the right hand side relation. Thus the left hand side is fixed at 16MB and the right hand side relation size increases from 1MB to 16MB. The results are shown in Figure 11. We can see that SC-1 has fixed overhead in all cases and almost parallel to SC-n, BR-n and BR-NC-n. This overhead (5.7%) is still much less than tuple-based rereading (178%, both at 16MB). R-n still exhibits the worst performance.

We found when we examined higher chunky skew levels, that SC-n had somewhat worse performance than BR-NC-n, due to the random writes to spool the cache, which are not necessary for the rereading algorithms. However, large chunky skew levels are rare in practice, because of the very large resulting relation size (approximating Cartesian product sizes).

As we expected, BR-S-n is better than BR-n since it avoids rereading (in our test case, the size of skewed data is less than the block size). It seemed that BR-S-n should have the best performance since it requires no rereads at all. However, in our results this does not hold. After examining the details of disk operation of BR-S-n and SC-n, we found that BR-S-n does indeed emit the least number of sequential reads. But it emits more random writes than SC-n. The elapsed time includes the time of writing the join result to the disk. Although the join results are the same for all the algorithms, the writing time is not the same. All the algorithms except for BR-S-n read a whole block of tuples into the memory at a time. BR-S-n reads part of a block into the memory when it encounters dual skew. Apparently BR-S-n executes more read operations. The more read operations, the higher probability for a read falling between two writes for the output results, forcing the second write to also seek. The expensive random write makes the cost of BR-S-n higher than SC-n.

Consider, however, the situation of using a separate disk for the output relation. In this particular case,



Algorithm	**Time** (sec)
R-n	8,014
SC-1	3,046
BR-n	2,917
BR-S-n	2,886
SC-n	2,881
BR-NC-n	2,881

Figure 11: LHS fixed at 16MB, with chunky skew

BR-S-n would be preferable, as the few additional read operations could not indirectly cause more seeks. In any case, the four best algorithms are all within a minute of each other, or 2% of the total time.

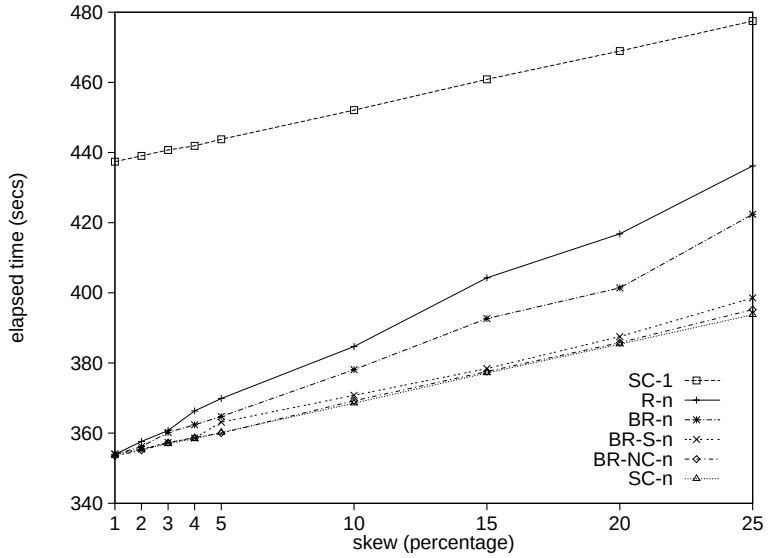
4.1.2 Smooth Skew

In this experiment, a series of relations were generated with a fixed size of 8MB and with increasing skew on the join attribute, from 1% to 25%. A relation has 1% smooth skew when 1% of the tuples in the relation have one duplicate value on the join attribute and 98% of the tuples have no duplicates. We examined self-joins to ensure that the LHS and RHS have the same degree of skew. The results are shown in Figure 12. Note that the y-axis starts at 340 sec, to emphasize the difference between the algorithms, which is less than that for chunky skew. At large skew, the difference between the slowest (SC-1) and fastest (SC-n) is 21% of the fastest time.

The graph shows SC-1 has the highest cost. This is due to one more scan on left hand side relation to merge multiple runs into one run. All the other algorithms proposed in this paper shows lower cost than R-n. The difference increases along with the increasing of skew percentage. The more skew in the relations, the higher probability that the skew appears at the boundary of blocks, and the more hiccups and thus disk reads for R-n. BR-n has less rereading than R-n due to its block-based rereading. As for BR-S-n, SC-n and BR-NC-n, there are at most two tuples in the cache at any time. The cache never overflows and there is no rereading. No extra I/O overhead is caused by smooth skew. Therefore, these three algorithms behave similarly and show the best performance. Again, the random writes in BR-S-n make its performance a little worse than SC-n.

4.1.3 No Skew

A critical question is how much extra cost our algorithms impose when there is no skew. In this experiment, we fixed the RHS size to 16MB and let the LHS size vary from 1MB to 16MB. All the relations have no skew. The results are shown in Figure 13. As before, SC-1 has higher cost than all the other algorithms. All the other algorithms have almost the same performance. Our data shows that BR-n and BR-S-n has only 0.0021% extra overhead, and SC-n and BR-NC-n have about 0.12% extra cost compared with R-n (which



Algorithm	**Time** (sec)
SC-1	477
R-n	436
BR-n	422
BR-S-n	399
BR-NC-n	395
SC-n	394

Figure 12: Fixed relation size (8MB) with smooth skew

is the traditional sort-merge join in the absence of skew). This is not difficult to explain. The only overhead of BR-n and BR-S-n is to test the prediction at the end of each block, which is a simple attribute compare operation. As for SC-n and BR-NC-n, they may need to add one tuple into the cache for each block, which is a tuple move operation. These overheads are all minor CPU-only costs (there are no additional I/O's in the absence of skew for any of the algorithms), and are extremely low.

4.2 Cache Size

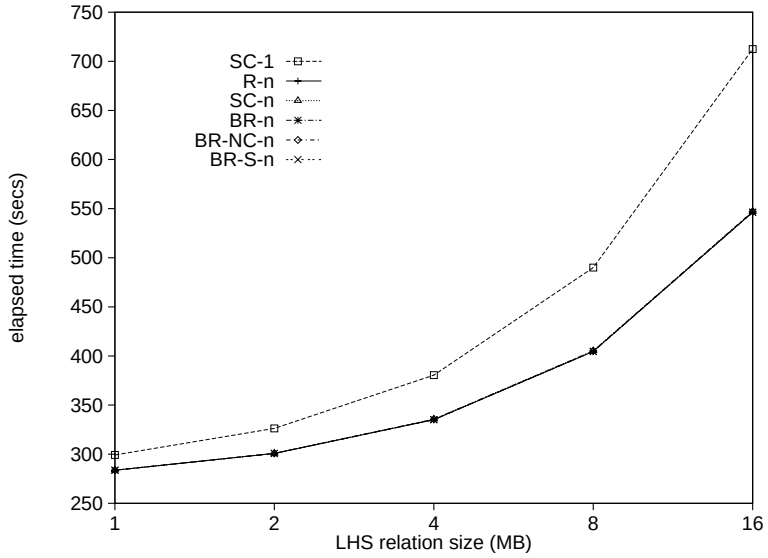
For all of the above experiments, we used a cache size of 32KB for the cache-based algorithms: SC-1, SC-n and BR-NC-n. The cache size is impacted only by the size of individual value packets, and so need be only as large as the biggest value packet.

For the smooth skew experiments, the largest value packet was two tuples, and so any cache will be large enough. For the chunky skew experiments, 1% skew represents a value packet of 10KB (80 tuples) for a 1MB LHS up to 160KB (1280 tuples) for a 16MB LHS. As such, it overflows at a LHS relation of 4MB, and indeed we see that in Figure 10. (The effect is small because there is only one such value packet.)

Some vendors (such as Oracle) now support automatic memory management. Each relational operator (join, sort, aggregation) can ask for corresponding memory according to the situation at run-time. With this feature, the join could use the maximum skew (which might be estimated from attribute statistics) to set an appropriate cache size. If the cache overflows, the operator can decide whether to increase the cache size (if the unexpectedly large value packet occurs early, and is likely to happen again) or spool the cache (if the large value packet occurs later, and is likely to be spurious).

4.3 Summary

The results of the experiments show that in all cases of skew, SC-n and BR-NC-n have the best performance. All the algorithms proposed in this paper perform much better than the traditional sort-merge join algorithm, R-n, under chunky skew. Most of them are also better than R-n under smooth skew except for SC-1. The effect of cache overflow and block-based rereading are almost the same under chunky skew since both cache and block share the space of the fixed main memory. SC-1 is not a good choice when no skew or small



Algorithm	**Time** (sec)
SC-1	712.41
BR-NC-n	547.09
SC-n	547.07
BR-S-n	546.41
BR-n	546.41
R-n	546.40

Figure 13: No skew

skew appears in the relations. SC-n, BR-n, BR-S-n, and BR-NC-n have almost the same performance as traditional sort-merge join in the absence of skew.

5 Band Join

We now consider a particular non-equijoin in which skew is more likely to happen: *band join* [DeWitt 91]. A band join between relations L and R on attributes $L.A$ and $R.B$ is a join in which the join condition can be written as $RL.A - c_1 \leq R.B \leq L.A + c_2$. Consider the query finding the salary of the employees from the Accounting department and the average salary of all employees that entered the company at about the same time. Assuming the unit of time is day and "about the same time" means a time difference less than 90 days, the query can be expressed in SQL as follows.

```
SELECT E1.Name, E1.salary, AVG(E2.salary)
FROM   Emp as E1, Emp as E2
WHERE  E1.Dept = 'Accounting'
   AND E2.start >= E1.start - 90 AND E2.start <= E1.start + 90
GROUP BY E1.Name
```

Such a query would be amenable to a band join, as the alternative would probably be nested loop. (We note in passing that temporal joins [Soo et al. 94] exhibit a very similar structure; much of the following also applies to temporal joins.)

Consider a sort-merge join implementation of this band join. For each tuple in `E1`, its value packet includes all the tuples in `E2` with the join value falling in the indicated range. This implies large (non-disjoint) value packets, and hence skew and in particular dual skew, is more likely to happen. As hiccups are expensive, this skew must be handled carefully.

5.1 Band Join Algorithms

The conventional join algorithms discussed in this paper are appropriate for band join, with two changes. First, the prediction rule should be changed, to: dual skew is present if the value of the last tuple in RHS

block falls within the band defined by the value of the last tuple in LHS block.

The second change is a more sophisticated purging policy for the algorithms with an auxiliary value-packet cache (BR-S-n, SC-1, SC-n, and BR-NC-n. In the equi-join algorithms, above, purging an existing value packet in the cache is easy. Because all the tuples in the value packet are point values, we simply clear the cache (both the in-memory and spooled portions). There is no overhead for this purging operation.

In a band join, because the RHS value packets are not disjoint, some of the tuples in the cache will be part of the next value packet. So it is necessary to only purge the beginning unqualified tuples (termed *garbage collecting the cache*), rather than the entire cache contents. For example, if our join condition is $L.A - c_1 \leq R.B \leq L.A + c_2$ and the current (LHS) join value changes from A to $A+1$, we need to remove all the $(A - c_1)$ tuples in the cache, and reorganize the cache. It is not clear to do this. If we purge the cache too often, this overhead can become significant. If we do not purge the cache, the cache will become larger and the cost for joining with the cache will become greater. Note though that the cache can be purged while joining it with the LHS, because the cache is sorted.

We modified the algorithms discussed in Section 3 to support band join. We eliminated from consideration SC-1, because the extra pass is not competitive; BR-NC-n was eliminated because it is too complex. We are left with the three most promising algorithms, SC-n, BR-S-n and BR-n, along with the simplest, R-n. Since they are band join algorithms, we call them BDSC-n, BDBR-S-n, BDBR-n and BDR-n respectively. For BDSC-n, the cache is garbage collected when tuples need to be added (this garbage collection can occur as the cache is scanned).

5.2 Experiments for Band Join

As in Section 4.1, we did experiments for band join algorithms on chunky skew, smooth skew and no skew. For chunky skew, the sizes of LHS and RHS relations are the same as in Section 4.1.1. The relations on both sides have 1% skew on their join attribute. One value of the join attribute repeats in the LHS relation and the number of duplicates is 1% of the total number of the tuples in the relation. In the RHS relation, the number of tuples with the join attribute value in the band defined by the skewed LHS value is also 1% of the total number of the tuples in the RHS relation. The results are shown in Figure 14.

For smooth skew, we use the same data as in Section 4.1.2. The constants defining the band are $c_1 = 0$ and $c_2 = 1$ respectively. Thus, the degree of skew is almost the same as in Section 4.1.2. The results are shown in Figure 15.

From these two plots, we see that the results are very similar to the results of equi-join experiments. For chunky skew, the costs of BDBR-S-n and BDSC-n are almost identical, and show that these two algorithms exhibit the best performance. BDR-n exhibits poor performance; the other three algorithms are much better. For smooth skew, all the four algorithms show the same relative performance as in equi-join. The new algorithms add virtually no extra cost to the traditional sort-merge join in the absence of skew. Specifically, BDSC-n has 0.03% extra cost, while BDBR-n and BRBR-S-n exhibit only 0.0007% extra cost in the absence of skew.

6 Conclusions

While skew has been investigated in detail for hash-join, there have been only general recommendations for how to handle skew in sort-merge join. In this paper, we subdivide intrinsic skew into (a) LHS skew, which presents no problem, (b) RHS skew, and (c) dual skew. We proposed several variants of sort-merge join that can accommodate all aspects of intrinsic skew: Reread with one run (R-1), Reread with multiple runs (R-n), Block-based Reread with one run (BR-1) and for multiple runs (BR-n), Block-based Reread with Smart use of memory (BR-S-n), Spooled Cache for skewed data with one more pass on the LHS for one run

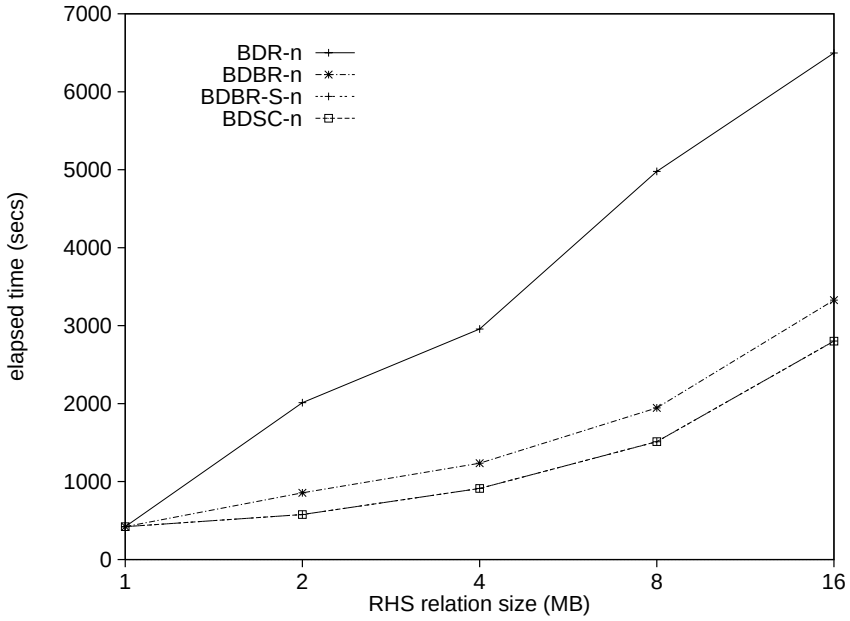


Figure 14: Different relation size with chunky skew of 1% for band join

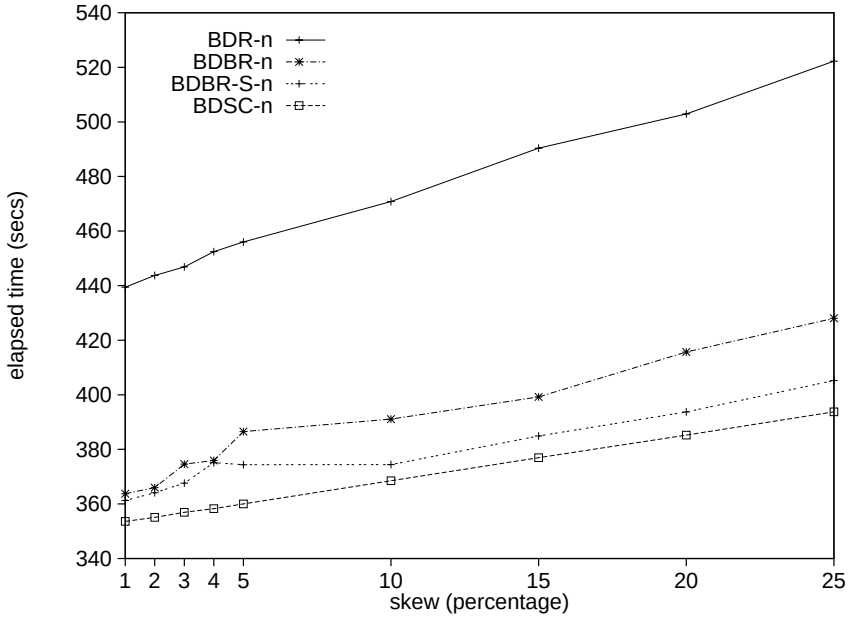


Figure 15: Fixed relation size (8MB) with smooth skew for band join

(SC-1) and for multiple runs (SC-n), and Block-based Reread with a Non-spooled Cache on multiple runs (BR-NC-n).

We tried these algorithms out on a variety of relation sizes, for smooth skew, chunky skew, and no skew, and with different percentages of skew. All of the algorithms proposed here perform much better than the traditional sort-merge algorithm, R-n, in the presence of chunky skew. SC-n, BR-n, BR-S-n and BR-NC-n have almost the same performance as traditional sort-merge in the absence of skew.

We also looked at four variants that deal with skew for band join. As before, the performance of BDR-n (the traditional sort-merge join) is much worse than the new algorithms. All the three new algorithms also did well in the absence of skew.

Taking all of these experiments into account, SC-n has slightly better performance, and of the four competitive algorithms (the other three being BR-n BR-S-n and BR-NC-n), SC-n is the easiest to implement. Hence, we recommend that the existing sort-merge join be replaced with SC-n, which exhibits strikingly better performance in the presence of skew, both for conventional and band joins, and exhibits virtually identical performance as traditional sort-merge join in the absence of skew. Concerning the cache size, our recommendation is to use a small cache, unless it is known that large value packets are prevalent.

7 Acknowledgements

This work was supported in part by NSF Grant EIA-0080123 and by a grant from Boeing Corporation.

References

[DeWitt 91] David J. DeWitt, Jeffrey F. Naughton and Donovan A. Schneider, "An Evaluation of Non-Equijoin Algorithms," *Proceedings of the International Conference on Very Large Databases*, Barcelona, Spain, pp. 443–452, September, 1991.

[DeWitt et al. 92] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider and S. Seshadri, "Practical Skew Handling in Parallel Joins," *Proceedings of the International Conference on Very Large Databases*, Li-Yan Yuan (ed.), Vancouver, British Columbia, Canada, pp. 27–40, August, 1992.

[Garcia-Molina et al. 93] Hector Garcia-Molina, Jeffrey D. Ullman and Jennifer Widom, **Database System Implementation**, Prentice Hall Publishers, 1999.

[Graefe 93] Goetz Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys* 25(2):73–170, June 1993.

[Graefe 94] Goetz Graefe, "Sort-Merge-Join: An Idea Whose Time Has(h) Passed?" in *Proceedings of the IEEE International Conference on Data Engineering*, Houston, TX, pp. 406–417, February, 1994.

[Graefe et al. 94] Goetz Graefe, Ann Linville and Leonard D. Shapiro, "Sort vs. Hash Revisited," *IEEE Transactions on Knowledge and Data Engineering* 6(6):934–944, December, 1994

[Gunadhi and Segev 90] Himawan Gunadhi and Arie Segev, "A Framework for Query Optimization in Temporal Databases," in *Proceedings of the the International Conference on Statistical and Scientific Database Management*, Zbigniew Michalewicz (ed.), pp. 131–147, Charlotte, NC, April, 1990.

[Hua and Lee 91] Kien A. Hua and Chiang Lee, "Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning," in *Proceedings of the International Conference on Very Large Data Bases*, Guy M. Lohman, Amlcar Sernadas and Rafael Camps (eds.), Barcelona, Catalonia, Spain, pp. 525–535, September, 1991.

[Kitsuregawa et al. 89] Masaru Kitsuregawa, Masaya Nakayama and Mikio Takagi, "The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method" in *Proceedings of the International Conference on Very Large Data Bases*, Peter M. G. Apers and Gio Wiederhold (eds), pp. 257–266, Amsterdam, The Netherlands, August, 1989.

[Kline & Soo 98] Roger N. Kline and Michael D. Soo, "The TIMEIT Temporal Database Testbed," 1998. `www.cs.auc.dk/TimeCenter/software.htm`.

[Kooi 80] Robert P. Kooi, "The Optimization of Queries in Relational Databases" Ph.D. thesis, Case Western Reserve University, 1980.

[Leung and Muntz 92] T. Y. Cliff Leung and Richard R. Muntz, "Generalized Data Stream Indexing and Temporal Query Processing," in *Second International Workshop on Research Issues in Data Engineering: Transaction and Query Processing*, February, 1992.

[Mishra and Eich 92] Priti Mishra and Margaret H. Eich, "Join Processing in Relational Databases" *ACM Computing Surveys*, 24(1):63–113, March 1992.

[Nakayama et al. 88] Masaya Nakayama, Masaru Kitsuregawa and Mikio Takagi, "Hash-partitioned Join Method Using Dynamic Destaging Strategy," in *Proceedings of the International Conference on Very Large Data Bases*, Francois Bancilhon and David J. DeWitt (eds), pp. 469–478, Los Angeles, CA, August, 1988.

[Segev and Gunadhi 89] Arie Segev and Himawan Gunadhi, "Event-Join Optimization in Temporal Relational Databases" in *Proceedings of the International Conference on Very Large Databases*, Peter M. G. Apers and Gio Wiederhold (eds), pp. 205–215, Amsterdam, The Netherlands, August, 1989.

[Selinger et al. 79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie and Thomas G. Price, "Access Path Selection in a Relational Database Management System," in *Proceedings of the SIGMOD International Conference on Data Management*, Philip A. Bernstein (ed.), Boston, Massachusetts, pp. 23–34, May, 1979.

[Soo et al. 94] Michael D. Soo, Richard T. Snodgrass and C. S. Jensen, "Efficient Evaluation of the Valid-Time Natural Join," in *Proceedings of the International Conference on Data Engineering*, Houston, TX, February, 1994, pp. 282–292.

[Walton et al. 91] Christopher B. Walton, Alfred G. Dale, and Roy M. Jenevein, "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins," in *Proceedings of the International Conference on Very Large Data Bases*, Guy M. Lohman, Amlcar Sernadas and Rafael Camps (eds.), Barcelona, Catalonia, Spain, pp. 537–548, September, 1991.

[Zhang et al. 01] Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy Lohman, "On Supporting Containment Queries in Relational Database Management Systems," in *Proceedings of the SIGMOD International Conference on Management of Data*, pp. 425–436, Santa Barbara, May, 2001.

[Zurek 96] Thomas Zurek, **Parallel Temporal Nested-Loop Joins** Ph.D. Dissertation, Dept. of Computer Science, Edinburgh University, 1996.