Skew Handling Techniques in Sort-Merge Join

Wei Li — Oracle Corporation — weili@us.oracle.com

Dengfeng Gao, Richard T. Snodgrass — Department of Computer Science, University of Arizona — {dgao,rts}@cs.arizona.edu

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. However, for sort-merge join, even a small amount of skew present in realistic data can result in a significant performance hit on a commercial DBMS. This paper examines the negative ramifications of skew in sort-merge join and proposes several refinements 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 thus suitable for replacing existing sort-merge implementations. We also show how sort-merge band 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 [18], 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 shown that their costs differ mostly by percentages [6, 7]. 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 [20].

Partition skew is of concern in hash-based join. In the first step of hash join, some buckets may contain more tuples than other buckets due to an interaction between the distribution of attribute values and the hashing function itself. 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. Several papers have proposed ways to deal with partition skew in hash-based join [3, 9, 10, 17, 20].

Intrinsic skew occurs when attributes are not distributed uniformly; it has also been called *attribute value skew* [20]. 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 relations. 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* [19] and *temporal Cartesian product* [22], and in *multi-predicate merge join* [21] and $\mathcal{EE}$-Join and $\mathcal{EA}$-Join [14] proposed for queries on XML data, exacerbate the problem.

The general advice is to use sort-merge join in the presence of significant intrinsic skew, because bucket overflow in hash join is so expensive. However, we are aware of no paper either on 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. While the sort-merge implementations in commercial systems yield correct results, we'll see shortly that even a small amount of intrinsic skew present in realistic data can result in a significant performance hit.

In this paper we provide a variety of algorithms that correctly and efficiently 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). We first demonstrate the high cost of intrinsic skew on a commercial system on actual data; the rest of this paper shows how to reduce almost completely this performance penalty. Section 3 identifies the three problems that skew presents to sort-merge join and shows how two of these problems can be solved. Section 4 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 6 shows how the algorithms perform for band joins [2], in which substantial skew is invariably present. Finally, Section 7 concludes with a recommended replacement for the traditional sort-merge join algorithm.

2. THE COST OF INTRINSIC SKEW

It may be surprising that anything new can be said about the venerable sort-merge join. The problem of 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 up 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 4.3). Such approaches can impose a significant performance penalty, as we now illustrate.

We used an actual data set from the University Information System (UIS), a major research university’s personnel database. Specifically, we used the `Incumbents` table, which includes information on job assignments for University employees. The size of the table is 7.8MB and has the following schema.

```
Incumbents (SSN, PCN, start_date, end_date,
       pay_hourly_rate, incum_fte, obj_code,
       track_code)
```

Two queries on this table are shown in Figure 1. *Q1* pairs rows that have the same key. *Q2* pairs those employees that have the same `pay_hourly_rate` during the same time period. The primary key of `Incumbents` is (`SSN`, `PCN`, `start_date`), which implies that there is no skew present in *Q1*. However, intrinsic skew does impact *Q2*. 96.4% tuples have unique `pay_hourly_rate` values. The remaining 3.6% tuples have duplicate values for this attribute. This small amount of skew turns out to decrease the performance of the join significantly.

The DBMS uses sort-merge join for these two queries. We measured both the elapsed time and the number of physical reads performed by just the merge step of each query, by separating out the time and number of physical reads for sorting. We varied the memory allocated for merging from the default size (64KB) defined by this DBMS to a much larger 1MB.

While the execution time to sort depends heavily on the size of main memory, the merge phase should be linear in the cardinality of the resulting table and independent of the size of main memory. The result size of *Q2* is only 1.5% larger than the result size of *Q1*, and so *Q1* and *Q2* should behave similarly. However, as shown in Figure 2, the performance of the merge step for this commercial DBMS degrades greatly in the presence of skew, especially when the size of memory is small. For 64KB of memory (the default size), the count of physical

Q1:
```
select i1.SSN, i1.pay_hourly_rate, i2.SSN
from   incumbents i1, incumbents i2
where  i1.SSN = i2.SSN
and    i1.PCN = i2.PCN
and    i1.start_date = i2.start_date
```

(a)

Q2:
```
select i1.SSN, i1.pay_hourly_rate, i2.SSN
from incumbents i1, incumbents i2
where i1.pay_hourly_rate=i2.pay_hourly_rate
and ((i1.start_date >= i2.start_date
      and i1.end_date <= i2.end_date)
  or (i1.start_date > i2.start_date
      and i1.end_date > i2.end_date
      and i1.start_date < i2.end_date)
  or (i1.start_date < i2.start_date
      and i1.end_date < i2.end_date
      and i2.start_date < i1.end_date)
  or (i1.start_date <= i2.start_date
      and i1.end_date >= i2.end_date))
```

(b)

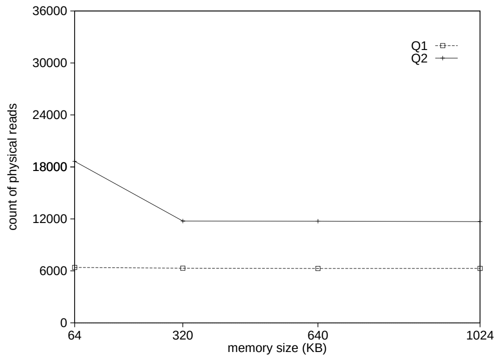
Figure 1: Two queries used to examine the performance of commercial DBMS

reads of *Q2* is almost three times that of *Q1*. The difference between the elapsed time is even larger, five times, due to the prevalence of the random reads that are involved when skew is improperly handled, as we will discuss shortly.

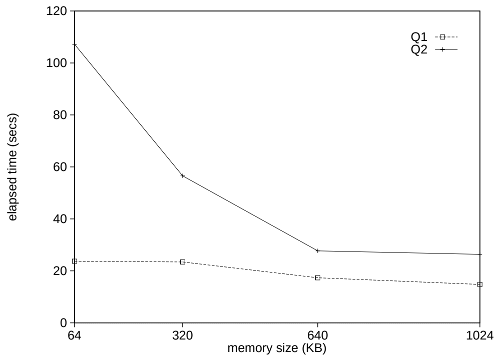
We’ve heard informally that some commercial sort-merge algorithms are variants of the R-1 approach we introduce below; our simulation studies for R-1 exhibit performance results similar to that shown in Figure 2. However, as we show, R-1 and its multi-run variant R-n are not competitive in performance 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 algorithms in the literature that are based on sort-merge join and could enhance the performance of commercial sort-merge join implementations on realistic data.

3. 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 [16]. We 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



(a) Count of physical reads



(b) Elapsed time

Figure 2: Performance of merge in sort-merge join in a commercial DBMS

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 3(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 simply the presence of multiple tuples in L or R with identical values for the equijoin attribute. These tuples, collectively called a *value packet* [5, 12] for each such value, are contiguous in the input relations after they are sorted. So an equivalent definition of skew is the presence of a value packet containing more than one tuple. The traditional algorithm must be modified to backtrack, yielding the algorithm in Figure 3(b). This algorithm effectively applies nested loop (cf. the nested repeats) on the tuples in 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 3(b) is *tuple-oriented*: the input relations are treated as in-memory arrays of tuples. Join implementations are almost always *block-based*, in which a block (or several blocks) 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 tuples) whenever a pointer is indexed out of the in-memory block, as shown with new code for *advance* in Figure 3(c) 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 buffer boundary. (A value packet entirely contained in a buffer presents no problem.) These three cases are shown in Figure 4. 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 buffer.

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 [5]. 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 [16]. This necessitates several passes over the same set of tuples of the inner relation. So whenever a duplicate LHS value is encountered, 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 the book by Garcia-Molina et al., which we will consider further in Section 4.4.

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 buffer boundary is encountered in the outer loop. The subsequent blocks of the LHS are 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 buffer boundary is encountered within the inner loop. In the presence of RHS skew, subsequent blocks of the RHS are 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 buffer 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 buffer, so subsequent tuples in the left-hand value packet will only be joined with the second portion of the right-hand value packet.

```
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 $L[pL](EA) \neq R[pR2](EA)$
            or $pR2 = R$.length
        advance $pL$
      else if $L[pL](EA) > R[pR](EA)$
        advance $pR$
      else                      // $L[pL](EA) < R[pR](EA)$
        advance $pL$
```

(b)

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

(c)

Figure 3: Traditional sort-merge join algorithms, original (a) and accommodating skew (b), and rendering sort-merge join block-based (c)

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 join, which is always possible due to the commutativity of join, though that swap may have implications to 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 that is present argued against adopting hash join.

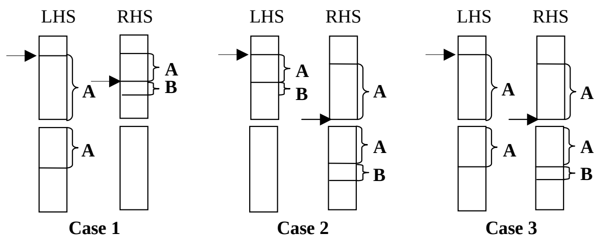


Figure 4: Types of skew in sort-merge join

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. Our algorithms with multiple runs described in the following sections are based on this optimization. This impacts how dual skew is accommodated.

In our implementation, each run is accorded one buffer in main memory, with the size of this *run buffer* dependent on the amount of available main memory and the size of the relation. In our description, the term ”buffer” denotes ”run buffer”. To simplify the implementation, multiple blocks that can fit in one buffer are read into memory at one time unless we explicitly mention otherwise.

The equijoin predicate may consist of multiple equality conditions that require the value of several columns to be equal. The query optimizer might choose to merge the relations on only some of the equality columns, with the remaining equality columns in the residual predicate processed in the inner loop of the merge algorithm. One possible reason is the presence of convenient indexes in the database on these particular columns. Our algorithms apply in this situation especially because skew is likely to appear on the merge columns. From this point on, the term ”equijoin attribute(s)” denotes the attribute(s) on which the input relation is sorted and merged, separate from the residual predicate.

4. CONTENDING WITH SKEWED DATA

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

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

R-1 is a simple extension of Figure 3(b)+Figure 3(c), in which a block 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. The down side is that an extra pass is needed to produce a single run for the right hand side.

To accommodate RHS and dual skew, the R-1 algorithm reads blocks when necessary (when a pointer is incremented past the end of a buffer) 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 ($pR2 \leftarrow pR$).

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.) The difference between single run and multiple runs is, when multiple runs are deployed, the final merge of the sort phase is done during joining phase. With multiple runs, the rereading process becomes more complex. For each RHS run, we record the backup pointer ($pR2$ in Figure 3(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) and the subsequent blocks. 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.

If the inner relation is an intermediate result, and thus it is not easy to rewind within it, the intermediate result can be written out in a temporary file or can be materialized as a B-tree index. The algorithms proposed in this section and in the following sections can be applied to either of the cases. Another variant of join utilizes iterators to supply tuples from left or right arguments [5]. Although the implementation of iterators generally doesn't allow rewinding to the previous tuples, in Section 4.4 we will see an approach which works directly with an iterator.

4.2 Block-based Reread (BR-1 and 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 3 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 buffer matches (for the equijoin attribute) that of the value of the last tuple of the right buffer.

In BR-1, shown in Figure 5, the LHS and RHS in-memory buffers are denoted by BL and BR, respectively, with a buffer size of B tuples. pL, pR and $pR2$ range from 1 to B, pointing into the main-memory buffer 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 block.

BR-n avoids the last run merge by storing information about the state of each LHS run. The idea is the same as BR-1; the algorithm is similar but more complex. Since a value packet can be distributed across buffers in multiple runs in both sides, to make sure each LHS block joins with each RHS block exactly once, the algorithm has to remember with which block each LHS run has finished joining.

A note on the implementation: the prediction rule requires that we look at the last record in the buffer. 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 buffer, the offset of the last record can be recorded in each buffer when writing out the run in the sort phase.

BR-n handles hiccups on a buffer basis, across many runs. Even with this optimization, hiccups are still quite expensive, as we'll see in Section 5. The next four algorithms attempt to avoid hiccups in the presence of skew.

4.3 Block-based Reread with Smart Use of Memory (BR-S-n)

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

BR-S-n handles hiccups on a single-block basis. As illustrated in Figure 6, when the end of the RHS buffer is encountered and dual skew is detected, we clearly know that all the blocks preceding the current value packet have been joined and need not be kept in the memory. Thus, these blocks can be discarded and their space can be used to hold the tuples in the current value packet that resides in the successive blocks. This involves shifting the current value packet, which resides at the very bottom of the buffer 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 buffer and thus avoid rereading altogether.

In the extreme situation that the size of skewed data exceeds the buffer size, the reread can't be avoided. But we are careful to reread from the block that contains the starting tuple of the value packet, so that the blocks preceding the most recent value packet are not reread: only the most recent value packet and subsequent blocks need to be reread. One potential disadvantage of BR-S-n is it partially loads a buffer into memory instead of loading blocks for a whole buffer at one time. Therefore, it may change some sequential reads to random reads.

4.4 Spooled Cache (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 hold the tuples in a given value packet. Subsequent blocks of both the LHS and RHS value

```
Block-based Sort-Merge Join :
  Sort relation $L$ on the attribute $EA$
  Sort relation $R$ on the attribute $EA$
  read blocks for one buffer of $L$ into $BL$
  $pL \leftarrow 1$
  read blocks for one buffer of $R$ into $BR$
  $pR \leftarrow 1$
  repeat until finished with $L$
    $pR2 \leftarrow \perp$
    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 $pR2 = \perp$
          $pR2 \leftarrow pR$       // remember start of
                                    // value packet
        if $SP(BL[pL], BR[pR])$
          output($BL[pL] \circ BR[pR]$)
        if $pR \neq B$              // end of RHS buffer
          advance $pR$
        else
          $pred \leftarrow (BL[B](EA) = BR[B](EA))$
          repeat until no matching tuple exists in RHS
                                    // finish off value packet
            join all the matching tuples in $BL$ and $BR$
            read the next blocks for one buffer of RHS
          if $pred$                 // dual skew
            advance $pL$            // to next buffer of LHS
            $pR \leftarrow pR2$     // restore RHS pointer
          else                      // RHS skew only
            advance $pL$            // to new value packet
            advance $pR$            // to new value packet
            $pR2 \leftarrow \perp$
    if $pR2 \neq \perp$
      $pR \leftarrow pR2$
    advance $pL$
```

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

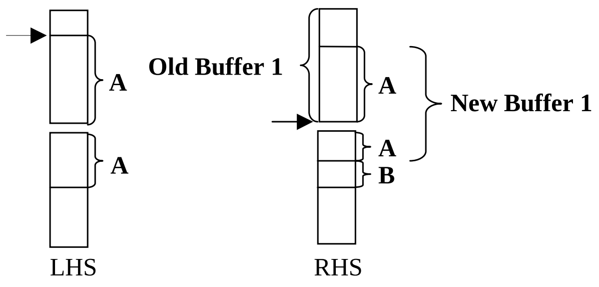


Figure 6: Smart use of block space

packets are read in, replacing blocks already scanned. If the value packet still doesn't fit in main memory, nested loop on the tuples in the value packet is required.

As they note, this algorithm is difficult to generalize to multiple runs. However, no details were presented, nor 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 reserve a 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 yet been completely joined 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 within 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 prevent the cache from overflowing. Here, we adopt the first approach; in Section 4.6, 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. In both sections, the spooled cache avoids rereading the previous blocks, and therefore works with an iterator.

We need to handle both RHS and dual skew with the cache (as before, 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 joining with future tuples, 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 in the cache. Otherwise, we avoid this overhead by joining all the RHS skewed data with all the corresponding LHS tuples.

2. *Join before caching 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.

$$\boxed{\textbf{Invariant:}\ \text{tuples in the cache have been joined with all tuples in previously-read LHS buffers}}$$

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

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

- After the buffer 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 in a new LHS buffer, we should first join all the tuples in the cache with the tuples in the just-read LHS buffer.

```
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 the blocks for one buffer of $L$ into $BL$
$pL \leftarrow 1$
read the blocks for one buffer of $R$ into $BR$
$pR \leftarrow 1$
repeat until finished reading $L$
  $pR2 \leftarrow \bot$
  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 $BL[pL](EA)$
        $pR2 \leftarrow pR$        // remember start of
                             // value packet
      if $SP(BL[pL], BR[pR])$
        output($BL[pL] \circ BR[pR]$)
      if $pR \neq B$
        $pR \leftarrow pR + 1$
      else                   // end of $R$ buffer
        purge the join-condition cache
        join all the corresponding tuples in $BL$ and $BR$
        if prediction on $BL$
          move skewed tuples into cache
        advance $pR$
        $pR2 \leftarrow pR$        // modify start position of
                             // value packet
  if $pR2 \neq \bot$
    $pR \leftarrow pR2$            // restore RHS reading pointer
  advance $pL$
  if $pL = 1$                // new $L$ buffer
    join the join-condition cache with all the tuples
    in $L$'s new buffer
```

Figure 7: Spooled cache algorithm, SC-1

The SC-1 algorithm is shown in Figure 7.

A similar approach was used in the tree-merge structure join algorithm [1], in which the nodes in a current sweep (analogous to our value packet) are stored in a temporary SHORE file, whose pages are written to and from disk by the SHORE buffer manager. Our appoach uses the prediction rule to determine whether to put tuples into the cache and exerts more careful control over the cache.

4.5 Spooled Cache 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 before the merge step of the join. 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 buffers in Run2 inconveniently occurs within a value packet), which means that it has just encountered dual skew.

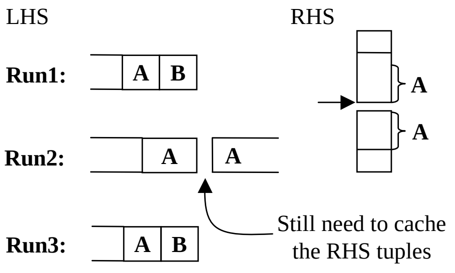


Figure 8: Several runs versus only one run?

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**: indicates 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 need 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 buffer (corresponding to the last statement inside the inner loop in Figure 7), the prediction rule is used to check whether we should move the RHS tuples into the cache. If there is no need to expand the cache, tuples in current RHS run are joined with tuples in each LHS run. Otherwise, tuples in current RHS run are only joined with tuples in **complete** LHS runs and the applicable RHS tuples are moved into the cache.

When any LHS run reaches the end of one buffer (corresponding to the last **if** statement in Figure 7), the algorithm checks the status of this run. If this run is a **complete** run, just load the subsequent blocks for this run and change this run's status to **pending**. If instead this run is a **pending** run, the algorithm joins all the **pending** LHS runs with the tuples in the cache and changes the status of all the LHS runs to **complete**. Then, the subsequent blocks are loaded and the run's status is changed to **pending**.

In this algorithm, purging the cache becomes a little more complex than in SC-1, where we simply set the cache to empty 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 buffer 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.

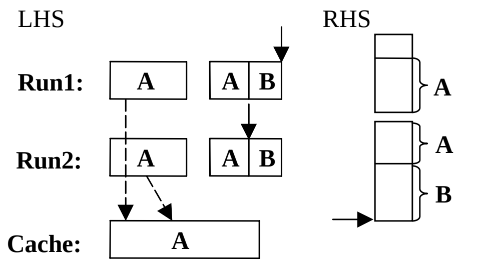


Figure 9: Purging the cache with multiple runs

4.6 Block-based Reread with a Non-Spooled Cache (BR-NC-n)

Algorithm R-n (cf. Section 4.1) 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 a new hot point. This is the most complex algorithm of the ones we propose.

5. EVALUATION AND COMPARISON

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 noncompetitive. 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 TIMEIT system [11], a software package supporting the prototyping of database components. Some parameters are fixed for all the experiments. They are shown in Table 1(a). The cache size for those algorithms that use a cache was set at 3% of the available main memory. In all test cases, the generated relations were randomly ordered, and the join algorithms were run with a cold main memory.

Parameter	Value
memory size	1MB/16MB
cache size	32KB/512KB
output buffer size	32KB
block size	1KB
tuple size	128 bytes
join attribute	4 bytes

(a)

Metric	Conversion
sequential I/O cost	1 msec
random I/O cost	10 msec
attribute compare	20 nsec
pointer swap	60 nsec
tuple move	640 nsec

(b)

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

TIMEIT collects a variety of metrics, shown in Table 1(b); both main memory operations and disk I/O operations were measured. TIMEIT 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.

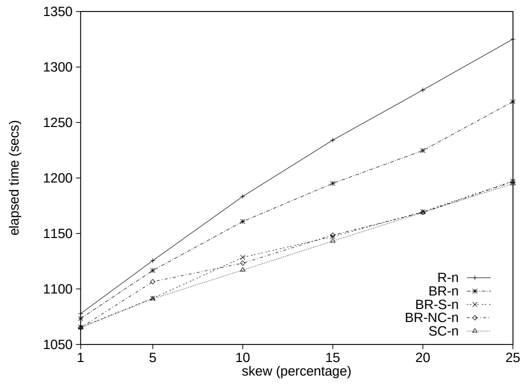
5.1 Experiments

Data skew is the presence of a repeated value in the equi-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 effecting a very large value set. We examine the performance of the various algorithms under these two kinds of skew.

5.1.1 Smooth Skew

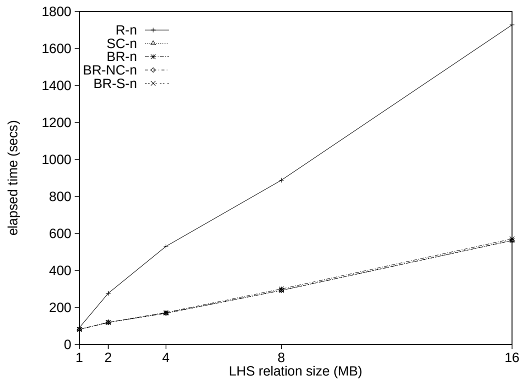
In this experiment, we fixed the memory size (16MB) and cache size (512KB). A series of relations were generated with a fixed size of 128MB and with increasing skew on the join attribute, from 1% to 25%. A relation has 1% smooth skew when 1% of the tuples in the relation have one duplicate value on the 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 10. Note that the y-axis starts at 1050 seconds to emphasize the difference between the algorithms, which is less than that for chunky skew. At large skew, the difference between the slowest (R-n) and fastest (SC-n) is 11% of the fastest time.

The graph shows the performance of the algorithms fall into



Algorithm	**Time** (sec)
R-n	1325.1
BR-n	1268.8
BR-S-n	1197.2
BR-NC-n	1196.6
SC-n	1194.8

Figure 10: Fixed relation size (128MB) with smooth skew



Algorithm	**Time** (sec)
R-n	1728
BR-n	571
SC-n	563
BR-NC-n	562
BR-S-n	561

Figure 11: Varying relation size with chunky skew of 1%

three groups. R-n has the highest cost. BR-n exhibits a lower cost than R-n but is worse than all the other algorithms, which constitute the third group. The difference between these groups increases along with the increasing skew percentage. The more skew in the relations, the higher probability that the skew appears at the boundary of buffers 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 value set 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.

5.1.2 Chunky Skew

At first, we tried to use large memory and larger relations. However, for one test case, the program for R-n didn't complete after 30 hours. The test case joined a 256MB relation with a 32MB relation using 16MB memory. Both of the relations have 1% chunky skew. 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 number of skewed tuples in the LHS relation is about 20000. Since skew appears in RHS relation, the skewed tuple must be distributed at least in two buffers. That means at least the blocks in two buffers were read 20000 times. In this case, the buffer size is about 0.8MB. This indicates 32GB extra reads and joins, which is 1000 times the size of the RHS relation(!). We conclude that the performance of tuple-based reread degrades significantly with the increase of the number of skewed tuples. A large data set with a small percentage of chunky skew renders tuple-based reread impractical.

Therefore, we decide to use a small memory (1MB) with small data set to compare our algorithm and tuple-based reread. Since the memory is small, we chose a smaller cache (32KB) than in smooth skew (that is, 3% of main memory). 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. The results are shown in Figure 11.

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 three times slower than the other algorithms. The number of skewed tuples determines the number of hiccups. We counted the number of hiccups in all the experiments. According to our data, the number of hiccups in R-n surpasses 13000 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 the cache to 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.

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 buffer size). BR-S-n has the similar performance to SC-n and BR-NC-n, because BR-S-n essentially uses the RHS run buffer as a cache when skew is present. Therefore, BR-S-n emits the least number of reads. However, if one buffer can't hold the value set, SC-n with a larger cache will perform better than BR-S-n.

5.1.3 No Skew

A critical question is how much extra cost our algorithms impose when there is no skew present. In this experiment, we use the same parameters as for smooth skew. We fixed the RHS size at 128MB and let the LHS size vary from 16MB to 128MB. All the relations have no skew. All the algorithms have almost the same performance for each size of relation.

The results for the smallest (16MB with 128MB) and the largest relation size (128MB with 128MB) are shown in Table 2. Table 2 does not include the result data for BR-S-n because it has exactly the same cost as BR-n. Our data shows that the algorithms we proposed have at most 0.002% extra overhead compared with R-n (which 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 buffer, which is a simple attribute compare operation. As for SC-n, BR-NC-n and BR-S-n, they may need to add one tuple into the cache or shift one tuple for each buffer, 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.

LHS	Elapsed Time (sec)			
size	R-n	BR-n	BR-NC-n	SC-n
16MB	447.5724	447.5733	447.5729	447.5913
128MB	805.5714	805.5787	805.5877	805.5785

Table 2: No Skew

5.2 Cache Size

For the above experiments, we used a 3% cache size: a 32KB cache for 1MB memory and a 512KB cache for 16MB memory for the cache-based algorithms, 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 11. (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 more memory according to the situation encountered at run-time. With this feature, the join algorithm 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). For small-footprint applications, it is best to use only a small cache and spool that cache when necessary.

5.3 Summary

The results of the experiments show that in all cases of skew, SC-n, BR-NC-n and BR-S-n have the best performance. All the algorithms proposed in this paper perform much better than the traditional sort-merge join algorithm, R-n. All the algorithms we proposed have almost identical performance as traditional sort-merge join in the absence of skew. The effect of cache overflow and block-based rereading are almost the same under chunky skew since both cache and buffer share the space of the fixed main memory.

Among the new algorithms we proposed, R-1 and R-n retain the sort order of the outer input in the output result. The block-based algorithms and the cache-based algorithms might change the order of the outer input, due to the complexity of multiple runs; however, the result remains sorted on the join attribute. In all the cases, if there is only one run in each of the input relations, the order of the input relations will be retained completely in the result.

Now consider a multiway join instead of two-way join, for example, a three-way join. If the three-way join merges the relations on the same columns, a spooled cache approach can be applied since the result is sorted on the merge columns.

6. BAND JOIN

We now consider a particular non-equijoin: *band join* [2]. 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 $L.A - c_1 \leq R.B \leq L.A + c_2$. Skew is more likely to happen in band join. 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 [19] 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.

6.1 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 identify dual skew when the value of the last tuple in RHS buffer falls within the band defined by the value of the last tuple in LHS buffer.

The second change is a more sophisticated purging policy for the algorithms with an auxiliary value-packet cache (SC-n and BR-NC-n). In the equi-join algorithms discussed in Section 4, 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

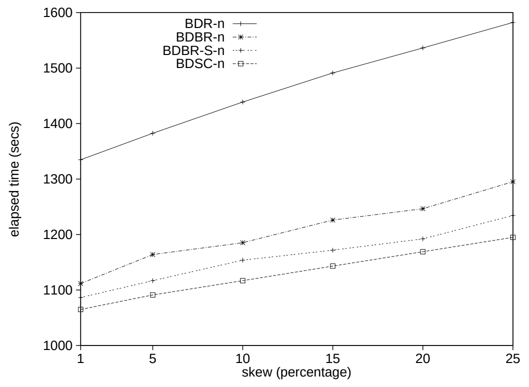
(LHS) join value changes from A to $A + 1$, we need to remove all the tuples in the cache with join attribute value less than $(A + 1 - c_1)$ and reorganize the cache. A design decision is when to garbage collect the cache. If we purge the cache too often, this overhead can become significant. If we do not purge the cache, the cache will become larger and the cost for joining with the cache will become greater.

We modified the algorithms discussed in Section 4 to support band join. We eliminated from consideration BR-NC-n 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).

6.2 Experiments for Band Join

As in Section 5.1, we did experiments for band join algorithms on chunky skew, smooth skew and no skew. We discuss the results for smooth skew and no skew.

For smooth skew, we use the same data and same parameters as in Section 5.1.1. 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 5.1.1. The results are shown in Figure 12. From the plot, we see that the results are very similar to the results of equi-join experiment. The results for chunky skew can be found in the full version of the paper [15]. All the four algorithms show the same relative performance as in equi-join.



Algorithm	**Time** (sec)
BDR-n	1582
BDBR-n	1295
BDBR-S-n	1234
BDSC-n	1195

Figure 12: Fixed relation size (128MB) with smooth skew for band join

The new algorithms add virtually no extra cost to the traditional sort-merge join in the absence of skew. Specifically, we used the same parameters and data set as in Section 5.1.3. For the largest relation size, BDSC-n has 0.005% extra cost, while BDBR-n and BDBR-S-n exhibit only 0.002% extra cost in the absence of skew (Table 3).

Algorithm	R-n	BR-n	BR-S-n	SC-n
Time (sec)	805.4957	805.5099	805.5099	805.5369

Table 3: Band join without skew

7. 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. We showed that even a small amount of dual skew can have a significant detrimental effect on the performance of a commercial DBMS on realistic data. We proposed several variants of sort-merge join that can accommodate 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 (SC-1) and for multiple runs (SC-n) and Block-based Reread with a Non-spooled Cache on multiple runs (BR-NC-n).

We experimented with these algorithms on a variety of relation sizes, for smooth skew, chunky skew and with varying percentages of skew. All of the algorithms proposed here perform much better than the traditional sort-merge algorithm, R-1 and its multi-run variant 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 three of the new algorithms also did well in the absence of skew.

If it is known a priori that there is no skew on the right hand side relation, for example, if no duplicates exist in the sense of equality join, then the simpler join algorithm without backup can be used in the situation. However, the overhead of all the algorithms in the presence of no skew is so small that we doubt that having a separate join algorithm is justifiable. 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) 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, for both conventional and band joins, and exhibits virtually identical performance as traditional sort-merge join in the absence of skew.

We know some vendors use single-run algorithms rather than multi-run algorithms for sort-merge join. Those vendors can benefit from SC-1 which is the single-run counterpart for SC-n. Concerning the cache size, our recommendation is to use a cache with the same size as the (run) buffer.

8. ACKNOWLEDGMENTS

This work was supported in part by NSF Grants EIA-0080123 and IIS-0100436 and by grants from Amazon.com and the Boeing Corporation. We would like to thank Goetz Graefe for his extensive comments on a previous version of this paper.

9. REFERENCES

[1] Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava and Yuqing Wu, “structural Joins: A Primitive for Efficient XML Query Pattern Matching,” *Proceedings of the International Conference on Data Engineering*, San Jose, CA, February, 2002.

[2] 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*, Guy M. Lohman, Am Alcar Sernadas and Rafael Camps (eds.), Barcelona, Spain, pp. 443–452, September, 1991.

[3] 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.

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

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

[6] 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.

[7] 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

[8] 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.

[9] 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.

[10] 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.

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

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

[13] T. Y. Cliff Leung and Richard R. Muntz, “Generalized Data Stream Indexing and Temporal Query Processing,” in *International Workshop on Research Issues in Data Engineering: Transaction and Query Processing*, Philip S. Yu (ed.), pp. 124–131, Tempe, AZ, February, 1992.

[14] Quanzhong Li and Bongki Moon, “Indexing and Querying XML Data for Regular Path Expressions,” in *Proceedings of the International Conference on Very Large Databases*, Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, Richard T. Snodgrass (eds.), pp. 361–370, Rome, September 2001.

[15] Wei Li, Dengfeng Gao and Richard T. Snodgrass, “Skew Handling Techniques in Sort-Merge Join,” 2001. `http://www.cs.auc.dk/research/DP/tdb/TimeCenter/TimeCenterPublications/TR-62.pdf`.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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*, Timos K. Sellis, Sharad Mehrotra (eds.), pp. 425–436, Santa Barbara, May, 2001.

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