Diag-Join: An Opportunistic Join Algorithm for 1:N Relationships

Sven Helmer  Till Westmann  Guido Moerkotte
Diag-Join: An Opportunistic Join Algorithm for 1:N Relationships

Sven Helmer    Till Westmann    Guido Moerkotte

Lehrstuhl für Praktische Informatik III
Universität Mannheim
68131 Mannheim
Germany

helmer|westmann|moer@pi3.informatik.uni-mannheim.de
Phone: +49 (621) 292 5582  Fax: +49 (621) 292 3394

Abstract

Time of creation is one of the predominant (often implicit) clustering strategies found not only in Data Warehouse systems: line items are created together with their corresponding order, objects are created together with their subparts and so on. The newly created data is then appended to the existing data. We present a new join algorithm, called Diag-Join, which exploits time-of-creation clustering. The performance evaluation reveals its superiority over standard join algorithms like nested-loop join and GRACE hash join. We also present an analytical cost model for Diag-Join.

1 Introduction

During the evaluation of queries in Data Warehouses, relations containing millions or even billions of tuples need to be joined. Joins involving fact tables are very costly operations. Evidently, fast join algorithms are very important in this environment.

The main strategy to lower join cost is to filter out many non-qualifying tuples beforehand. Bit-vector indexing is predominantly used for this purpose, like in O’Neil’s and Graefe’s multi-table join [31]. However, it may not always be possible to filter out a significant number of tuples. The join attribute may also take on many different values, leading to huge bit-vectors, so that the overhead of filtering may not pay off. We have asked ourselves, if properties of relations exist, that can be exploited somehow during a join operation. During our analysis we made the following observations. When inserting new tuples into a Data Warehouse, those tuples are usually appended to existing relations [20, 24]. Therefore time of creation is the predominant—though often implicit—clustering strategy. Another important observation was that in the context of Data warehousing relations are typically joined on foreign keys [20, 24]. Backed by these observations, we developed a join algorithm—called Diag-Join— which takes advantage of these facts. It exploits time-of-creation clustering for 1:n relationships.

Let us illustrate these two points by an example taken from [24]. All companies selling products have to ship these products to their customers, Hence, the process of shipping goods plays an important role. Assume that in the Data Warehouse of such a company
a central fact table Shipments exists, that contains the data on all deliveries made. In a
dimensional table CustomerOrders we store information on all orders that the company
received. See Figure 1 for an illustration. Soon after appending an order from a customer,
we expect the corresponding tuples to be added to Shipments, resulting in clustering by
time-of-creation.

<table>
<thead>
<tr>
<th>Product Key</th>
<th>Price</th>
<th>Shipment Date</th>
<th>ShipMode</th>
<th>OrderNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>24.00</td>
<td>10/12/96</td>
<td>Mail</td>
<td>K-323</td>
</tr>
<tr>
<td>234</td>
<td>35.00</td>
<td>10/13/96</td>
<td>Air</td>
<td>K-323</td>
</tr>
<tr>
<td>655</td>
<td>97.00</td>
<td>11/13/96</td>
<td>Air</td>
<td>K-323</td>
</tr>
<tr>
<td>534</td>
<td>1298.00</td>
<td>11/23/96</td>
<td>Truck</td>
<td>K-326</td>
</tr>
<tr>
<td>239</td>
<td>20.00</td>
<td>12/10/96</td>
<td>Air</td>
<td>K-351</td>
</tr>
<tr>
<td>978</td>
<td>1000.00</td>
<td>12/18/96</td>
<td>Rail</td>
<td>K-351</td>
</tr>
<tr>
<td>174</td>
<td>35000.00</td>
<td>12/20/96</td>
<td>Ship</td>
<td>K-351</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OrderNo</th>
<th>CustomerID</th>
<th>RetailPrice</th>
<th>OrderDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-323</td>
<td>1943</td>
<td>1751.00</td>
<td>11/20/96</td>
</tr>
<tr>
<td>K-326</td>
<td>432</td>
<td>1751.00</td>
<td>11/20/96</td>
</tr>
<tr>
<td>K-351</td>
<td>129</td>
<td>45020.00</td>
<td>12/02/96</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 1: The relations Shipments and CustomerOrders

The Diag-Join exploits this clustering. In essence, Diag-Join is a sort-merge join
without the sort phase. An important difference, however, is that the merge phase of
Diag-Join does not assume that the tuples of either relation are sorted on the join attributes.
Instead, it relies on the physical order created by the (implicit) time-of-creation clustering strategy. More specifically, Diag-Join joins the two tables by scanning them
simultaneously. The scan on the outer relation proceeds by moving a sliding window of
adjustable size over the relation. Only within this window we search for join partners for
the inner relation. A special mechanism takes care of those tuples of the inner relation
for which no join partner could be found in the window. They are called mishits. Though
simple, this idea proves to be very effective. There are, however, some subtleties that are
addressed later on. These are the buffer management, the window size, the organization
of the window, and the sliding speed of the window.

Diag-Join has two advantages over other join algorithms:

- Even if the relations do not fit into main memory, in many cases Diag-Join will be
  able to avoid the creation of large temporary files, unlike the sort-merge join [1], the
  hybrid hash join [6, 35], and the GRACE hash join [10, 35].

- Contrary to other join algorithms, output tuples can be produced right away without
  a painful interruption of the query evaluation pipeline.

The rest of the paper is organized as follows. Section 2 covers related work. We
present the Diag-Join algorithm in Section 3. Section 4 contains performance evaluations
and comparisons with other join algorithms. Section 5 concludes the paper.

2 Related Work

Since the invention of relational database systems, tremendous effort has been undertaken
in order to develop efficient join algorithms. Starting from a simple nested-loop join
algorithm, the first improvement was the introduction of the merge join [1]. Later, the hash join [2, 6] and its improvements [21, 25, 30, 37] became alternatives to the merge join. (For overviews see [29, 35] and for a comparison between the sort-merge and hash joins see [12, 13].) A lot of effort has also been spent on parallelizing join algorithms based on sorting [9, 27, 28, 33] and hashing [5, 10, 34].

All of these algorithms concentrate on simple join predicates based on the comparison of two atomic values. Predominant is the work on equi-joins, i.e., where the join predicate is based on the equality of atomic values. Only a few articles deal with special issues like non-equi joins [8], non-equi joins in conjunction with aggregate functions [4], and pointer-based joins [7, 36]. An area where more complex join predicates occur is that of spatial database systems. Here, special algorithms to support spatial joins have been developed [3, 14, 19, 26, 32]. Another special join algorithm has been developed for joining objects on set-valued attributes [18].

Another important research area is the development of index structures that allow to accelerate the evaluation of joins [16, 22, 23, 31, 39, 40]. However, if there is no selection prior to a join or the selections exhibit a high selectivity value (i.e., many output tuples are produced), the performance gain of these algorithms is limited. This is also true for bitmap join indices [31], that were developed especially for Data Warehouse environments. Hence, we only incorporated standard join algorithms in our performance benchmarks.

3 The Diag-Join

The first subsection briefly summarizes some preliminaries and notations used throughout the rest of the paper. We then present a basic version of the Diag-Join explaining the principle of the algorithm. We proceed by giving an advanced version of the algorithm illuminating implementation details. The last subsection contains a cost model.

3.1 Preliminaries

For the rest of the paper we use the symbols depicted in Table 1. Given two relations \( R_1 \) and \( R_N \) to be joined, we assume that \( R_1 \) contains the key \( \kappa \), that \( R_N \) has as foreign key attribute(s). That is, a \( 1:n \) relationship exists between \( R_1 \) and \( R_N \). \(|R_x|\) denotes the cardinality (in number of tuples) of a relation \( R_x \) (with \( x \in \{1, N\} \)), while \(||R_x||\) stands for the size of \( R_x \) in pages. We further assume, that the tuples in each relation are (implicitly) numbered by their physical occurrence. The \( j \)-th tuple in \( R_x \) is denoted by \( R_x[j] \) with \( 1 \leq j \leq |R_x| \).

Let us assume that a tuple of \( R_1 \) and all matching tuples in \( R_N \) are created in the same transaction and are written to disk at the same time. We can easily figure out the physical position of the joining tuple in \( R_1 \) for a given tuple in \( R_N \). We call this situation “perfect” clustering by time-of-creation. In the special case of a \( 1:n \) relationships, i.e., every tuple in \( R_N \) joins exactly with one tuple from \( R_1 \), we expect for each tuple \( R_N[j] \) to find the matching tuple in \( R_1 \) at position \( \left\lfloor \frac{j}{|R_N|/|R_1|} \right\rfloor \). If the number of join partners of each tuple in \( R_1 \) varies, the calculated position is only an approximation. Figure 2 illustrates a perfect situation. On the x-axis we have the positions of the tuples in \( R_N \), on the y-axis the expected positions of their join partners in \( R_1 \). Here, each tuple in \( R_1 \)
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>(smaller) relation to be joined</td>
</tr>
<tr>
<td>$\kappa$</td>
<td>key of relation $R_1$</td>
</tr>
<tr>
<td>$R_N$</td>
<td>(larger) relation to be joined (with foreign key $\kappa$)</td>
</tr>
<tr>
<td>$</td>
<td>R_x</td>
</tr>
<tr>
<td>$</td>
<td>R_x</td>
</tr>
<tr>
<td>$R_x[j]$</td>
<td>tuple at position $j$ in relation $R_x$, $1 \leq j \leq</td>
</tr>
<tr>
<td>$t$</td>
<td>an arbitrary tuple</td>
</tr>
<tr>
<td>$m_b$</td>
<td>size of buffer/window in number of tuples</td>
</tr>
<tr>
<td>$m_p$</td>
<td>size of buffer/window in number of pages</td>
</tr>
<tr>
<td>$l$</td>
<td>size of array of hash tables</td>
</tr>
<tr>
<td>$p$</td>
<td>hash table size in pages ($= \frac{m_p}{l}$)</td>
</tr>
<tr>
<td>$selPred$</td>
<td>selection predicate</td>
</tr>
</tbody>
</table>

Table 1: Used symbols

joins with exactly two tuples from $R_N$. Hence, the join partner of $R_N[5]$ is $R_1[3]$, because $\left\lfloor \frac{5}{\frac{3}{4}} \right\rfloor = 3$.

![Figure 2: Expected positions of matching tuples](image)
**Diag-Join**($R_1$, $R_N$, m_t)

```c
{
    /* phase 1 */
    ratio = $|R_N| / |R_1|$;
    curTup = m_t/2;
    fill buffer with $R_1[1]$ to $R_1[curTup]$;
    for(j = 1; j <= $|R_N|$; j++)
    {
        if(tuple t in buffer matches $R_N[j]$)
        {
            join t with $R_N[j]$;
            output result;
        } else
        {
            write $R_N[j]$ to tmpfile;
        }
        if(j % ratio == 0)
        {
            curTup++;
            if(space left in buffer)
            {
                add $R_1[curTup]$ to buffer;
            } else
            {
                replace tuple with lowest position with $R_1[curTup]$;
            }
        }
    }
    /* phase 2 */
    join $R_1$ with tmpfile using standard join algorithm;
}
```

**Figure 3:** Basic Diag-Join algorithm
3.2 Basic Diag-Join

If the tuples in the relations are perfectly clustered, then a simple merge phase suffices
to join the two relations. However, in reality this is not always the case. There may be
some exceptions, because the number of join partners for each tuple in \( R_1 \) varies, the
tuples are not inserted simultaneously into \( R_1 \) and \( R_N \), or they are reorganized later (e.g.,
deletion of tuples, insertion of additional tuples, replacements). Hence we do not just look
at one tuple of \( R_1 \) at a time, but hold \( m_t \) tuples—those in the vicinity of the expected
position—in a buffer. We call the part of \( R_1 \) held in the buffer a window on \( R_1 \).

The basic Diag-Join algorithm works as follows. We initialize the window with
\[ \left\lceil \frac{m_t}{2} \right\rceil \]
tuples from \( R_1[1] \) to \( R_1[\left\lceil \frac{m_t}{2} \right\rceil] \). We expect the matching tuple for \( R_N[1] \) to be at \( R_1[1] \) or
in the range from \( R_1[\left\lceil \frac{m_t}{2} \right\rceil] \) to \( R_1[\left\lceil \frac{m_t}{2} \right\rceil] \). Since there are no negative positions in \( R_1 \),
this part is cut off. Then \( R_N \) is scanned sequentially starting with \( R_N[1] \). No buffering
is applied to \( R_N \), except for the current tuple. For every tuple \( R_N[j] \) we search
the window for a matching tuple from \( R_1 \). If the lookup is successful (we call this a hit), we
immediately produce an output tuple. If the lookup fails (called mishit), then \( R_N[j] \) is
written into a temporary file. Whenever \( |R_N|/|R_1| \) tuples from \( R_N \) have been processed,
we add the next tuple from \( R_1 \) to the window. If there is no free space left in the window,
we replace the tuple with the lowest position. When we have finished scanning \( R_N \), we
join the tuples in the temporary file (which should be much smaller than \(|R_N|\) with
\( R_1 \) using some standard join algorithm. Figure 3 gives a summary of the basic Diag-Join
algorithm.

Before presenting a more elaborate version of Diag-Join, let us briefly highlight some
problems of the basic version. First, the algorithm is not very efficient, because it uses
a tuple-oriented buffer, while most DBMSs use page-oriented structures. Second, the
organization of the window is also crucial for the efficiency and needs to be discussed.
Third, the algorithm only works on base relation, i.e. no selections prior to the join are
possible. We resolve these problems in the next section.

3.3 Advanced Diag-Join

We kept the algorithm in the last section very simple, because we intended to illustrate
the basic principle of the algorithm. The implementation details are presented in this
section.

We change from a tuple-oriented buffer to a page-oriented buffer. We do not read
single tuples into the window, but all tuples on the next \( p \) pages. As a consequence, if
the window buffer is full we replace \( p \) pages. Reading the tuples blockwise is much more
efficient. We call \( p \) the step size of Diag-Join. Obviously, we replace tuples in the window
whenever \( p \cdot |R_N|/|R_1| \) pages have been scanned in \( R_N \).

Searching the window sequentially for matching tuples is too expensive, therefore we
use hash tables to look up join partners in the window. There are two alternatives. We
can use one large hash table with a size of \( m_p \) pages or an array of \( l \) hash tables with a
size of \( \frac{m_t}{l} \) pages each. Using only a single hash table is disadvantageous. If we apply a
step size \( p \) equal to the window size \( m_p \), we also replace a part of the vicinity inserted
during the last step that is needed in the current step. If we apply a step size \( p \) smaller
than the window size \( m_p \), we must delete many tuples from the hash table individually.
Therefore we allocate an array of \( l \) hash tables. Each hash table has a size equal to \( \frac{m_p}{l} \).

We equate the hash table size with the step size, hence \( p = \frac{m_p}{l} \). Then in each step we free an entire hash table, which is much cheaper than deleting individual entries. Figure 4 depicts the window organization. The window size is six pages, organized into three chunks of two pages each. Therefore the step size is also equal to two pages. The broken lines indicate, how the pages are replaced when no free buffer space is left.

![Window organization for Diag-Join](image)

**Figure 4: Window organization for Diag-Join**

After describing the organization of the window let us now look at the algorithm. Sliding the window is done as follows. Whenever \( p \cdot ||R_N||/||R_1|| \) pages have been scanned in \( R_N \), the least recently loaded hash table is cleared and from \( R_1 \) the next \( p \) pages are loaded into this hash table. How do we look up matching tuples in the hash table array? First of all we search the middle table at position \( \left\lfloor \frac{t}{2} \right\rfloor \) in the array. If \( R_1 \) and \( R_N \) are perfectly clustered, we expect to find the matching tuple in this table. If we are not able to find it there, we search the table at position \( \left\lfloor \frac{t}{2} \right\rfloor + 1 \). On failure the tables at positions \( \left\lfloor \frac{t}{2} \right\rfloor - 1, \left\lfloor \frac{t}{2} \right\rfloor + 2, \left\lfloor \frac{t}{2} \right\rfloor - 2 \), and so on are searched. We call this technique *zig-zag search*. This is the best technique, when the deviation of the relations from perfect clustering can be described by a normal distribution. If the matching tuple is found, then we join the tuples immediately and output the result. Otherwise the tuple from \( R_N \) is written into a temporary file. To speed up the algorithm, we could hold the mishits in a main memory buffer. Only if this buffer overflows we begin to start writing the mishits to disk. Also we recommend to use an uneven number for \( l \), so that the searching range for the lookups is symmetrical.

We have to be careful when filtering out certain tuples with a selection prior to the join operation. If we feed the resulting tuples of the selection operators straight into the join operator, this may destroy the synchronization, i.e. we may slide the window at the wrong time. Therefore Diag-Join has to be synchronized with the scans on the base relations. We do this by using the Observer pattern described in [11]. The scan on the base relation \( R_N \) notifies Diag-Join, whenever \( p \cdot ||R_N||/||R_1|| \) pages have been scanned, so that Diag-Join slides the window at the right time.

The algorithm is summarized in Figure 5. Please note that the current middle table is not always at position \( \left\lfloor \frac{t}{2} \right\rfloor \), because we reuse the hash tables in the array.

7
Diag-Join(R_1, R_N, m_p, l, selPred)
{
    /* phase 1 */
    ratio = |R_N| / |R_1|;
    allocate array arr[l] of hash tables;
    fill arr[1] to arr[l/2] with tuples from R_1;
    do
    {
        get next tuple from R_N satisfying selPred;
        zig-zag search hash tables for matching tuple;
        if(matching tuple found)
        {
            join tuples;
            output results;
        }
        else
        {
            write R_N[j] to tmpBuf;
        }
        if(notified from scan on base relation R_N)
        {
            if(space left in arr)
            {
                load next p pages from R_1 into next free hash table;
            }
            else
            {
                clear least recently loaded hash table;
                load next p pages from R_1 into cleared hash table;
            }
        }
    } while (tuples from R_N remain);

    /* phase 2 */
    join R_1 with tmpBuf using standard join algorithm;
}

Figure 5: Advanced Diag-Join algorithm
### 3.4 Cost model

Our cost model for Diag-Join is based on the cost models presented in [17]. Additional parameters needed for the cost model are presented in Table 2. The cost \( C_{I/O} \) for transferring a set of \( ||R_x|| \) pages from disk to memory, or vice versa, through a buffer size \( B_x \) is given by

\[
C_{I/O} = (||R_x||, B_x) = \left( \frac{||R_x||}{B_x} \right) \cdot T_k + ||R_x|| \cdot T_t
\]

(1)

where \( T_k \) is composed of the sum of the average seek and latency time and \( T_t \) is the cost for transferring a page between disk and memory. The costs for Diag-Join consist of the costs for the first phase and the costs for the second phase.

\[
C_{DiAG}(R_1, R_N) = C_{Phase1} + C_{Phase2}
\]

(2)

In the first phase we have to read \( R_1 \) and \( R_N \), hash all tuples of \( R_1 \), look for matching tuples and join them or write the mishits to disk.

\[
C_{Phase1} = C_{Read \; R_1} + C_{CreateHash} + C_{Read \; R_N} + C_{Join} + C_{Write}
\]

(3)

The components of \( C_{Phase1} \) are defined as follows:

\[
C_{Read \; R_1} = C_{I/O}(||R_1||, m_p)
\]

(4)

\[
C_{CreateHash} = |R_1| \cdot T_c
\]

(5)

\[
C_{Read \; R_N} = C_{I/O}(||R_N||, 1)
\]

(6)

\[
C_{Join} = |R_N| \cdot T_j
\]

(7)

\[
C_{Write} = C_{I/O}(||tmpFile||, 1)
\]

(8)

The costs in the second phase depend on the join algorithm used. In our case we applied GRACE hash join in the second phase (for cost models of GRACE hash join see [15, 17]), hence

\[
C_{Phase2} = C_{GRACE}(R_1, tmpFile)
\]

(9)
### 3.5 Calculating the mishit probability

In this section we derive the formula for calculating the *mishit probability*, that is the probability that an arbitrary tuple from \( R_N \) turns out to be a mishit. With the help of this probability the size of the temporary file can be estimated. Table 3 summarizes the needed parameters.

As already mentioned, we assume that the derivation of the relations from perfect clustering can be described by a normal distribution. The normal distribution \( n(x, \mu, \sigma) \) with mean \( \mu \) and standard deviation \( \sigma \) is defined as follows.

\[
n(x, \mu, \sigma) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}
\]

(10)

We also need to know the probability that \( x \) is in the range between \( a \) and \( b \). This can be calculated by the distribution \( N(a, b, \mu, \sigma) \).

\[
N(a, b, \mu, \sigma) = \int_a^b n(x, \mu, \sigma) \, dx
\]

(11)

Let us illustrate what it means that the tuples are distributed normally among the relations. For the tuple \( R_N[i] \) at position \( i \) (\( 1 \leq i \leq |R_N| \)) in relation \( R_N \), we expect to find the matching tuple at position \( j = i \cdot \frac{|R_1|}{|R_N|} \) in relation \( R_1 \), if the relations are perfectly clustered. There may be some deviation, however, as indicated by the bell-shaped curve in Figure 6. The curve indicates the probability that the matching tuple can be found at position \( j = i \cdot \frac{|R_1|}{|R_N|} \) in \( R_1 \). \( w_{lo} \) and \( w_{hi} \) are the smallest and largest positions of the elements found in the window, respectively. The middle hash table in the window starts at position \( m_{lo} \) and ends at position \( m_{hi} \).

The probability that \( R_N[i] \) turns out to be a mishit is the probability that the matching tuple is not inside the window:

\[
Pr(R_N[i] \text{ is a mishit}) = 1 - N(w_{lo}, w_{hi}, j, \sigma)
\]

(12)

When scanning through \( R_N \) this probability changes, because \( j \) moves through the middle hash table from \( m_{lo} \) to \( m_{hi} \). Whenever \( j \) reaches \( m_{hi} \) the index slides down by the
specified step size and $j$ starts at the beginning of the next new middle hash table again. Hence, the average mishit probability $Pr_{avg}$ as $j$ moves through the middle hash table of a given window with boundaries $w_l$ and $w_h$ can be calculated by

$$Pr_{avg}(R_N[i]) \text{ is a mishit (for } \frac{|R_N|}{|R_1|} \cdot m_{lo} \leq i \leq \frac{|R_N|}{|R_1|} \cdot m_{hi}) = \sum_{j=m_{lo}}^{m_{hi}} \frac{1 - N(w_{lo}, w_{hi}, j, \sigma)}{m_{hi} - m_{lo}}$$  \hspace{1cm} (13)$$

As mentioned in Table 1 a window consists of $l$ hash tables and holds $m_t$ tuples. That means we can reformulate the mishit probability for the general case as follows:

$$Pr_{avg}(R_N[i]) \text{ is a mishit (for } 1 \leq i \leq |R_N|) = \sum_{j=h_t}^{h_t(\left\lceil \frac{i}{l} \right\rceil + 1)} \frac{1 - N(0, m_t - 1, j, \sigma)}{h_t}$$  \hspace{1cm} (14)$$

where $h_t = \frac{|R_1|d}{m_t}$ is the average number of tuples per hash table.

It would be interesting to know how large the window has to be chosen in order to achieve a mishit probability below a certain value $p_{accept}$, which is still acceptable.
That means, given $p_{\text{accept}}$, we want to calculate the corresponding value for $m_t$ using the expanded version of (14):

$$
\begin{align*}
    p_{\text{accept}} & \geq \left\lfloor \frac{1}{2} \right\rfloor + 1 + \sum_{j=\left\lfloor \frac{1}{2} \right\rfloor}^{m_t-1} \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-m_t)^2}{2\sigma^2}} dx \\
    & = \left(\frac{1}{2}\right)^{m_t} + 1 - \int_{0}^{m_t-1} \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-m_t)^2}{2\sigma^2}} dx
\end{align*}
$$

(15)

This formula is very impractical as the integral can only be estimated numerically and we still lack a way to determine $\sigma$ precisely. Therefore we recommend using histograms. Histograms can be built in a single scan through $R_1$ and $R_N$ with as large a buffer as possible. For each tuple in $R_N$ the absolute value of the difference between the expected position and the actual position of the matching tuple in $R_1$ is inserted into the corresponding bucket of the histogram. Mishits are counted separately. The resulting histogram (for an example see Figure 7) can be used to approximate the window size for a given probability $p_{\text{accept}}$.

![Histogram](image)

Figure 7: Histograms for measuring deviation from perfect clustering

4 Benchmarks

This section is composed of two parts. Within the first part we describe the benchmark environment and how the benchmarks were run. In the second part we present the results and analyze them.

4.1 Benchmark description

The benchmarks were executed on a lightly loaded UltraSparc 1 (143 MHz) with 288 MByte main memory running under Solaris 2.5.1. The data we worked with were generated for a TPC-D benchmark with a scaling factor of 1 [38]. We joined the relation $Order$ and $LineItem$ (see Figure 4 for the schemes). The relation $Order$ was sorted on the attribute $orderdate$, $LineItem$ was sorted on $shipdate$. Note that this does not result in
an ordering on the join attribute orderkey in the relations, but it nicely models clustering by time of creation.

The algorithm was implemented in C++ using the Sun C++ Compiler Version 4.1. It was integrated into our experimental Data Warehouse Management System AODB. We buffered one page of mishits in main memory. For the standard join algorithm in the second phase of Diag-Join we used GRACE hash join [10, 35].

In a first step we optimized some parameters of Diag Join, e.g. finding the optimal number of hash tables. Then we compared the total costs, CPU-based costs and I/O based costs of Diag Join with blockwise nested-loop join and GRACE hash join for different buffer sizes. We do not look at hybrid hash join, because for large relations relative to the size of main memory, GRACE hash join performs as well as hybrid hash join [17, 35]. As Table 5 shows, that summarizes the parameters for the benchmarks, the size of the buffer we used is at most \( \frac{1}{10} \) of the size of the relations. This is a realistic assumption for Data Warehouses in which huge relations can be found.

### 4.2 Benchmark results

#### 4.2.1 Tuning the Diag-Join algorithm

When joining relations with Diag-Join, we have to choose the right step size and window size. Two effects have to be considered. If we use a large number of hash tables (small step size), we avoid cutting off matching tuples in the vicinity of the expected positions. However, the more hash tables we use, the longer the zig-zag search will take. For small buffer sizes the step size plays almost no role, because the number of mishits caused by a large step size is small compared to the total number of mishits. For large
buffer sizes, however, the number of mishits is relatively small and the step size has a noticeable effect. The break-even points can be clearly seen on the left-hand side of Figure 8. Very small step sizes, on the other hand, do not improve the mishit ratio significantly. The run-time is dominated by the search time for the zig-zag search in this case. For our benchmarks we divided the window into five hash tables. This turned out to be a good compromise between optimizing the step size and the search time.

On the right-hand side of Figure 8 the percentage of mishits in the relation Lineitem is depicted. The results of these benchmarks are straightforward. The more buffer we allocate, the lower is the probability that a tuple from Lineitem will be a mishit, because the probability to find the matching tuple in a hash table increases. For large buffer sizes the effect of a large step size can be clearly seen as the percentage of mishits rises for a low number of hash tables.

Table 5: Parameters used for benchmarks

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Size</td>
<td>4 KByte</td>
</tr>
<tr>
<td>Size of Order</td>
<td>44,475 pages</td>
</tr>
<tr>
<td>Cardinality of Order</td>
<td>1,500,000 tuples</td>
</tr>
<tr>
<td>Size of Lineitem</td>
<td>189,635 pages</td>
</tr>
<tr>
<td>Cardinality of Lineitem</td>
<td>6,001,215 tuples</td>
</tr>
<tr>
<td>Window size for Diag-Join</td>
<td>300 - 4000 pages</td>
</tr>
<tr>
<td>Step size (Window size/5)</td>
<td>(1.17 MByte - 15.62 MByte)</td>
</tr>
<tr>
<td>Buffer size for Nested-loop join</td>
<td>300 - 4000 pages</td>
</tr>
<tr>
<td></td>
<td>(1.17 MByte - 15.62 MByte)</td>
</tr>
<tr>
<td>Buffer size for GRACE join</td>
<td>300 - 4000 pages</td>
</tr>
<tr>
<td></td>
<td>(1.17 MByte - 15.62 MByte)</td>
</tr>
</tbody>
</table>

Figure 8: Diag Join
4.2.2 Comparison with other join algorithms

In this section we compare Diag Join with blockwise nested-loop join and GRACE hash join. The results for total runtime of all algorithms for joining the relations Order and Lineitem on the attribute orderkey are shown in Figure 9.

Figure 9: Total runtime of join algorithms

Blockwise nested-loop join performs worst. This comes as no great surprise, because the ratio between the buffer size and the relations’ sizes is very unfavorable. For sufficiently large buffer sizes (>3000 pages or 6% of ||R1||) Diag Join outperforms GRACE hash join, because in this case all tuples are joined in the first phase of Diag Join and no additional phase for joining the mishits is needed. For very small buffer sizes (<1000 pages or 2% of ||R1||) GRACE hash join is only slightly faster than Diag Join. What are the reasons for this? The first phase of Diag Join has a relatively low overhead, but is still able to join a certain number of tuples (see Figure 10). This takes at least some of the load off

Figure 10: Total number of mishits
GRACE hash join in the second phase of Diag Join. The difference between the overhead for the first phase of Diag Join and the performance gain of GRACE hash join in the second phase is not as large as one might expect.

![Comparison of join algorithms](image)

**Figure 11:** CPU and I/O costs of join algorithms

Let us now have a look at the CPU-based costs of the join algorithms (see left-hand side of Figure 11). The more available memory we have, the lower the costs of the blockwise nested-loop join are. This is obvious as the number of necessary loops decreases with increasing buffer size. As long as it is sufficiently large, the size of the hash table directories is irrelevant for the CPU-based costs of GRACE hash join. The CPU-based costs for GRACE hash join are composed of the costs for hashing all tuples of *Order*, hashing all tuples of *LineItem*, hashing all tuples of *Order* again during the merge phase, and do *[LineItem]* lookups on this hash table. This leads to nearly constant costs. The CPU-based costs for Diag Join for the first phase are almost constant regardless of buffer size, because *Order* and *LineItem* are simply scanned (see Figure 12). The slight increase is caused by the costs for joining the tuples. The more available buffer there is in the first phase, the more tuples will find a join partner in this phase. (We did not write mishits to disk while measuring the CPU-based costs for the first phase.) The total decreasing CPU-based costs for Diag-Join are caused by falling costs of GRACE hash join in the second phase as the number of tuples in the temporary file steadily decreases.

The I/O-based costs are displayed on the right-hand side of Figure 11. For the blockwise nested-loop join we have the same behavior as for the CPU-based costs. The larger the buffer size, the smaller the number of loops, the lower the costs. For GRACE hash join the I/O-based costs decrease with increasing buffer size. Beyond a certain buffer size, however, the seek and latency time become small and the costs for transferring the data dominate. As *Order* and *LineItem* are always read twice and written once, more buffer does not change the transfer costs. Therefore the I/O-based costs level out. When allocating large buffers (≥ 3000 pages, which corresponds to about 6% of the size of *Order*) for Diag Join all we have to do is to read *Order* and *LineItem* once and we have finished. Hence we have small I/O-based costs in this case. For small buffers (< 3000 pages) all tuples of *Order* and *LineItem* are read once in the first phase. Additionally,
part of Lineitem is written into a temporary file, which is then joined with Order. When we decrease the buffer size, the temporary file will increase (because of a larger number of mishits) leading to higher join costs for GRACE hash join in the second phase.

4.3 Summary of Benchmarks

If we have a clustering of relations by time of creation, Diag Join performs very well (up to two and a half times faster than GRACE hash join and up to 28 times faster than blockwise nested-loop join). Diag Join needs sufficient memory (about 6% of $|R_i|$ in our benchmark) to achieve the best case, but even for small buffer sizes the performance is still satisfactory.

Obviously, when joining relations that are not clustered by time of creation, i.e. relations with randomly placed tuples, Diag Join will fail. In this case we expect a high rate of mishits as on average only $\frac{\text{buffer size}}{R_i}$ $R_N$ of the tuples in $R_N$ will find the matching tuple in the first phase.

5 Conclusion

We developed a join algorithm, called Diag-Join, for Data Warehouse environments in which joining very large relations is not unusual. We take advantage of the fact that in Data Warehouses new incoming data is appended at the end of relations, resulting in a clustering of the tuples by time of creation. When this is the case, often a single merge phase suffices to join these large relations. This results in lower join costs than the costs for any other join algorithm.

We implemented Diag-Join and integrated it into our experimental Data Warehouse Management System AODB. There we ran benchmarks based on the TPC-D relations Order and Lineitem. A careful analysis of the behavior of Diag-Join and the comparison to blockwise nested-loop join and GRACE hash join revealed the impressive performance
of our join algorithm. It ran up to two and a half times faster than GRACE hash join (the latter being on equal grounds with hybrid hash join in our case) and up to 28 times faster than blockwise nested-loop join. However, we recommend that Diag-Join should only be used for at least loosely clustered relations, because for non-clustered relations the results are less favorable, as we have the overhead of the first phase, but still almost all tuples have to be joined in the second phase by a standard join algorithm.

References


[34] D. Schneider and D. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 469–480, Brisbane, 1990.


