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

Sven Helmer Till Westmann Guido Moerkotte
helmer|westmann|moer@pi3.informatik.uni-mannheim.de

Fakultät für Mathematik und Informatik, University of Mannheim, Germany

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. If we are able to take advantage of time-of-creation clustering, then the performance evaluation reveals the superiority of Diag-Join over standard join algorithms like block-wise nested-loop join, GRACE hash join, and index nested-loop 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 [23]. 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 were wondering, 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 [13, 17]. 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 [13, 17]. 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 [17]. 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.

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

Proceedings of the 24th VLDB Conference
New York, USA, 1998

Shipments				
ProductKey	*Price*	*ShipDate*	*ShipMode*	*OrderNo*
123	24.00	10/12/96	Mail	K-323
234	35.00	10/13/96	Air	K-323
012	97.00	10/13/96	Air	K-323
635	1298.00	11/23/96	Truck	K-326
534	453.00	11/23/96	Truck	K-326
239	20.00	12/10/96	Air	K-351
978	10000.00	12/18/96	Rail	K-351
174	35000.00	12/20/96	Ship	K-351
...	...	...	...	...

CustomerOrders			
OrderNo	*CustomerID*	*TotalPrice*	*OrderDate*
K-323	1943	156.00	10/10/96
K-326	432	1751.00	11/20/96
K-351	129	45020.00	12/02/96
...	...	...	...

Figure 1: The relations *Shipments* and *CustomerOrders*

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. We also present a method which allows Diag-Join to join non-base relations (resulting from intermediate operations).

Diag-Join has two advantages over other join algorithms for appropriately clustered relations:

- 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 [4, 26], and the GRACE hash join [6, 26].
- 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 block-wise nested-loop join, GRACE hash join, and index nested-loop join (for a brief sketch on GRACE hash join, see Section 2). 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, 4] and its improvements [14, 18, 22, 27] became alternatives to the merge join. (For overviews see [21, 26] and for a comparison between the sort-merge and hash joins see [8, 9].) A lot of effort has also been spent on parallelizing join algorithms based on sorting [5, 19, 20, 24] and hashing [3, 6, 25].

For many applications hash-based join algorithms have proven to be superior. One of these algorithms is the GRACE hash join [6, 26]. As it plays a central role in our paper let us give a brief description of it. When joining two relations R and S, we partition them in a way such that the following two conditions are met. First, each of the partitions of the smaller one fits into main memory. Second, matching tuples are always found in corresponding partitions of the other relation. The algorithm performs the following steps (assuming R is the smaller relation):

1. Choose a hash function h so that it partitions R into r approximately equal-sized subsets. Allocate r output buffers.
2. Scan R, thereby hashing each tuple into the appropriate output buffer using h. If the output buffer is full, write it to disk. When finished with the scan, flush all buffers to disk.
3. Do the same for S.
4. For each of the r partitions read partition R_i into a main memory hash table. For each tuple in the corresponding partition S_i probe in this hash table to find a match.

Another important research area is the development of index structures that allow to accelerate the evaluation of joins [11, 15, 16, 23, 29, 30]. 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 bit-map join indices [23], that were developed especially for Data Warehouse environments. Hence, we only incorporated standard join algorithms in our performance benchmarks.

Symbol	*Definition*
R_1	(smaller) relation to be joined
κ	key of relation R_1
R_N	(larger) relation to be joined (with foreign key κ)
$\|R_x\|$	cardinality of relation R_x in number of tuples ($x \in \{1, N\}$)
$\|\|R_x\|\|$	size of relation R_x in number of pages
$R_x[i]$	tuple at position i in relation R_x, $1 \leq i \leq \|R_x\|$
t	an arbitrary tuple
m_t	size of buffer/window in number of tuples
m_p	size of buffer/window in number of pages
l	size of array of hash tables
p	hash table size in pages ($= \frac{m_p}{l}$)
$interOp(R_x)$	intermediate operator on R_x

Table 1: Used symbols

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 in subsection 3.2. We proceed by giving an advanced version of the algorithm illuminating implementation details in 3.3. We deal with the subtleties mentioned in the introduction. Further, we discuss how to join non-base relations (resulting from intermediate operations). The last two subsections contain a cost model and the derivation of formulas to calculate the *mishit probability* (i.e. the probability that a tuple turns out to be a mishit).

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 κ, which is foreign key of R_N. 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 i-th tuple in R_x is denoted by $R_x[i]$ with $1 \leq i \leq |R_x|$.

Let us assume that a tuple of R_1 and all matching tuples in R_N are created by 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 1:n relationships, i.e. every tuple in R_N joins exactly with one tuple from R_1, we expect for each tuple $R_N[i]$ to find the matching tuple in R_1 at position $\left\lceil \frac{i}{|R_N|/|R_1|} \right\rceil$. If the number of join partners of each tuple in R_1 varies, the calculated position is only an approximation. Figure 2 illuminates 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 joins with exactly two tuples from R_N. Hence, the join partner of $R_N[5]$ is $R_1[3]$, because $\left\lceil \frac{5}{8/4} \right\rceil = 3$. It is important to note that, even for perfect clustering, the relations will almost certainly not be sorted on the join attributes.

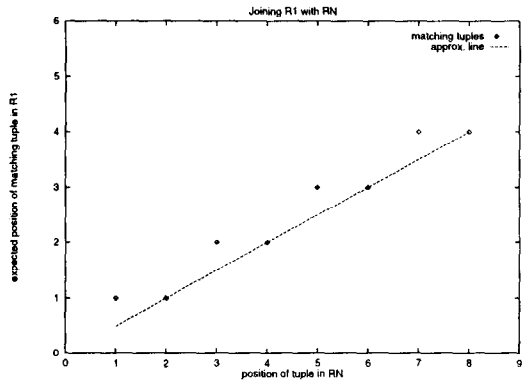

Figure 2: Expected positions of matching tuples

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 $\lceil \frac{m_t}{2} \rceil$ tuples from $R_1[1]$ to $R_1[\lceil \frac{m_t}{2} \rceil]$. We expect the matching tuple for $R_N[1]$ to be at $R_1[1]$ or in the range from $R_1[-\lfloor \frac{m_t}{2} \rfloor]$ to $R_1[\lceil \frac{m_t}{2} \rceil]$. Since there are no negative positions in R_1, the interval from $-\lfloor \frac{m_t}{2} \rfloor$ to 0 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 $R_N[i]$ 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

```
Diag-Join(R_1, R_N, m_t)
{
   /* phase 1 */

   ratio  = |R_N| / |R_1|;
   curTup = m_t/2;
   fill buffer with R_1[1] to R_1[curTup];
   for(i = 1; i <= |R_N|; i++)
   {
      if(tuple t in buffer matches R_N[i])
      {
         join t with R_N[i];
         output result;
      }
      else
      {
         write R_N[i] to tmpfile;
      }
      if(i % ratio == 0)
      {
         curTup++;
         if(buffer is full)
         {
            replace tuple with lowest position with R_1[curTup];
         }
      }
   }

   /* phase 2 */

   join R_1 with tmpfile using any standard join algorithm;
}
```

Figure 3: Basic Diag-Join algorithm

and go on to the next tuple in R_N. We can do this, because there can be at most one hit (1:n relationship). If the lookup fails (called *mishit*), then $R_N[i]$ 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 relations, e.g. 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.

3.3.1 Page-oriented buffer

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, which is much more efficient. We call p the *step size* of Diag-Join. As a consequence, we replace tuples in the window whenever $p \cdot ||R_N||/||R_1||$ pages have been scanned in R_N.

3.3.2 Hashing the window

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 $\frac{m_p}{l}$ 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.

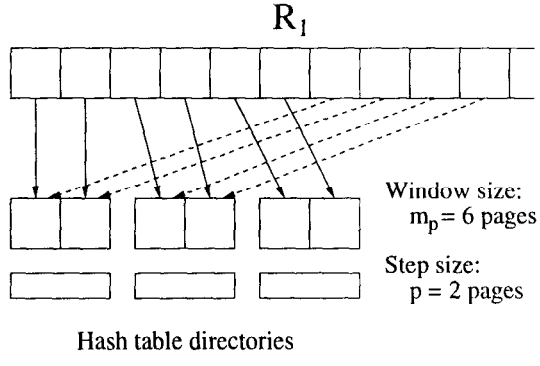

Figure 4: Window organization for Diag-Join

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. Then the next p pages from R_1 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 $\lceil \frac{l}{2} \rceil$ 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 $\lceil \frac{l}{2} \rceil + 1$. On failure the tables at positions $\lceil \frac{l}{2} \rceil - 1$, $\lceil \frac{l}{2} \rceil + 2$, $\lceil \frac{l}{2} \rceil - 2$, 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 (see Figure 7 in Section 3.5). 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 flush it to disk. We also recommend to use an odd number for l, so that the searching range for the lookups is symmetrical.

```
Diag-Join(R_1, interOp(R_N), m_p, l)
{
   /* 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
   {
      t_N = next tuple from interOp(R_N);
      zig-zag search hash tables for matching tuple;
      if(matching tuple found)
      {
         join tuples;
         output results;
      }
      else
      {
         write t_N to tmpBuf;
      }
      if(notified from access operator on base relation R_N)
      {
         if(all hash tables are full)
         {
            clear least recently loaded hash table;
            load next pages from R_1 into cleared hash table;
         }
      }
   } while (tuples from interOp(R_N) remain);

   /* phase 2 */

   join R_1 with tmpBuf using any standard join algorithm;
}
```

Figure 5: Advanced Diag-Join algorithm

3.3.3 Joining non-base relations

We have to take special care when joining non-base relations. If we feed tuples from intermediate operators (working on R_N) straight into a Diag-Join operator, this may destroy the synchronization, i.e. we may slide the window on R_1 incorrectly. We solve this problem by using the Observer pattern described in [7]. The intent of the Observer pattern (also known as publish-subscribe) is to notify all dependent objects $o_1, o_2, \ldots, o_n$ of a state change in an object s. For a description in C++ notation see Figure 6.

```
class Observer
{
   update(Subject*);
}


class Subject
{
   attach(Object*);
   detach(Object*);
   notify();
}
```

Figure 6: Observer Pattern

The methods *attach* and *detach* connect and disconnect objects to a subject object s. When s changes its state, it calls the method *notify* which in turn calls the method *update* of all observer objects currently attached. In our case the operator accessing the tuples from R_N (scan, index scan, etc.) notifies Diag-Join about the position within R_N from which the current tuples are fetched. Then Diag-Join is able to slide the window with the right speed or even skip some pages of R_1. Note that this technique allows any intermediate operator to occur between the scan on R_N and Diag-Join, as long as it preserves the relative order of the tuples. A similar technique can also be applied to handle intermediate operators between the scan on R_1 and Diag-Join. When loading a hash table during the advancement of the window, it is always filled completely. If an intermediate operator discards many tuples, the scan on R_1 may hurry ahead in order to fill the hash table. If the scan on R_1 notifies Diag-Join of the positions of the currently scanned tuples, Diag-Join will be able to recognize this case. As a consequence, Diag-Join will delay the window sliding on R_1 until the scan on R_N has caught up.

The algorithm is summarized in Figure 5. Please note that the current middle table is not always at position $\lceil \frac{l}{2} \rceil$, because we reuse the hash tables in the array.

3.4 Cost model

Our cost model for Diag-Join is based on the cost models presented in [12]. The 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 of size B_x is given by

$$C_{I/O}(||R_x||, B_x) = \left\lceil \frac{||R_x||}{B_x} \right\rceil \cdot T_k + ||R_x|| \cdot T_t \qquad (1)$$

Symbol	Definition
$C_{I/O}$	cost for transferring pages between disk and memory
B_x	arbitrary buffer
T_k	sum of average seek and latency time
T_t	time for transfer of one page
T_c	time for hashing a tuple
T_j	time for finding the join partner of a tuple

Table 2: Parameters for cost model

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 and the second phase.

$$C_{DIAG}(R_1, R_N) = C_{Phase1} + C_{Phase2} \quad (2)$$

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.

$$C_{Phase1} = C_{Read\ R_1} + C_{CreateHash} + C_{Read\ R_N} + C_{Join} + C_{Write} \quad (3)$$

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

$$C_{Read\ R_1} = C_{I/O}(||R_1||, p) \quad (4)$$
$$C_{CreateHash} = |R_1| \cdot T_c \quad (5)$$
$$C_{Read\ R_N} = C_{I/O}(||R_N||, 1) \quad (6)$$
$$C_{Join} = |R_N| \cdot T_j \quad (7)$$
$$C_{Write} = C_{I/O}(||tmpFile||, 1) \quad (8)$$

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 [10, 12]), hence

$$C_{Phase2} = C_{GRACE}(R_1, tmpFile) \quad (9)$$

Even though we present an estimation for normally distributed tuples in Section 3.5 approximating $||tmpFile||$ will not be a trivial task. As the assumption of normally distributed tuples is probably not valid for all applications we recommend the following procedure. During times of low workload (or an issued run-stat command) a shortened version of the first phase of Diag-Join is processed. This shortened version is used to determine $||tmpFile||$ without actually creating the temporary file or any result tuples.

The query optimizer of a DBMS needs to be supplied with the above cost model and its parameters

Symbol	Definition
$N(a, b, \mu, \sigma)$	normal distribution
$n(x, \mu, \sigma)$	density function of normal distribution
$j(i)$	expected position of matching tuple
$m_{lo}(i)$	start position of middle hash table
$m_{hi}(i)$	end position of middle hash table
$w_{lo}(i)$	start position of window ($w_{lo}(i) \leq m_{lo}(i)$)
$w_{hi}(i)$	end position of window ($m_{hi}(i) \leq w_{hi}(i)$)
h_t	average number of tuples per hash table

Table 3: Parameters for mishit probability

(especially an estimation of $||tmpFile||$) to enable it to make a decision about the application of Diag-Join. The costs for joining base relations can be approximated by using (2) without modifications. If order-preserving intermediate operators occur, the standard techniques of the optimizer to estimate the costs of complex queries have to be applied (e.g. calculating the cardinalities of the intermediate relations and the size of Diag-Join's temporary files (tmpFile) with the help of selectivities). At the moment we are working on formulas to estimate the "clusteredness" of an (intermediate) relation when applying different operators to it. Our goal is to enable the optimizer to approximate the costs of Diag-Join operators in more complex query-plans (involving operators that break the tuple ordering).

3.5 Calculating the mishit probability

In this section we derive a formula for calculating the *mishit probability*, that is the probability that an arbitrary tuple from R_N turns out to be a mishit. Table 3 summarizes the needed parameters.

With the help of this probability the size of the temporary file can be estimated:

$$|tmpFile| \approx Pr_{avg}(R_N[i]\text{is a mishit}) \cdot |R_N| \quad (10)$$

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 μ and standard deviation σ is defined as follows.

$$n(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{(x-\mu)^2}{2\sigma^2}} \quad (11)$$

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

$$N(a, b, \mu, \sigma) = \int_a^b n(x, \mu, \sigma) dx \quad (12)$$

Let us illustrate what we mean by normally distributed tuples. 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 $j(i) = \left\lceil i \cdot \frac{|R_1|}{|R_N|} \right\rceil$ 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 7. The curve indicates the probability that the matching tuple can be found at a certain position around $j(i)$ in R_1. The middle hash table in the window starts at position $m_{lo}(i)$ and ends at position $m_{hi}(i)$ (h_t is the average number of tuples per hash table):

$$m_{lo}(i) = \left(\left\lceil \frac{j(i)}{h_t} \right\rceil - 1 \right) \cdot h_t + 1 \quad (13)$$

$$m_{hi}(i) = \left(\left\lceil \frac{j(i)}{h_t} \right\rceil \right) \cdot h_t \quad (14)$$

$w_{lo}(i)$ and $w_{hi}(i)$ are the smallest and largest positions of the elements found in the window, respectively (we assume that l is odd):

$$w_{lo}(i) = m_{lo}(i) - \left\lfloor \frac{l}{2} \right\rfloor \cdot h_t \quad (15)$$

$$w_{hi}(i) = m_{hi}(i) + \left\lfloor \frac{l}{2} \right\rfloor \cdot h_t \quad (16)$$

Please note that for a better readability we have refrained from covering the special cases at the start and end of R_1.

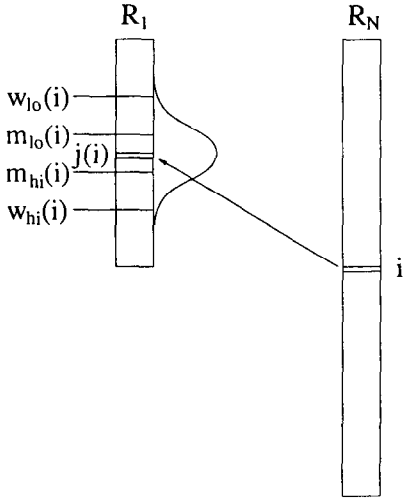

Figure 7: Normally distributed tuples

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

$$Pr(R_N[i] \text{ is a mishit}) = 1 - N(w_{lo}(i), w_{hi}(i), j(i), \sigma) \quad (17)$$

When scanning through R_N this probability changes, because $j(i)$ moves through the middle hash table from $m_{lo}(i)$ to $m_{hi}(i)$. Whenever $j(i)$ reaches $m_{hi}(i)$ the window slides down by the specified step size.

We are interested in attaining a mishit probability below a threshold value p_{accept}. This is tantamount to limiting the size of the temporary file. How large do we have to choose the window size m_t (and the step size h_t) to guarantee $Pr_{avg}(R_N[i] \text{ is a mishit}) \leq p_{accept}$? The mishit probabilities of the tuples in R_N repeat themselves for each window as $j(i)$ passes from $m_{lo}(i)$ to $m_{hi}(i)$. So the average mishit probability can be approximated by

$$Pr_{avg}(R_N[i] \text{ is a mishit}) = \sum_{j=\lfloor \frac{l}{2} \rfloor \cdot h_t + 1}^{\lceil \frac{l}{2} \rceil \cdot h_t + 1} \frac{1 - N(1, m_t, j, \sigma)}{h_t} \quad (18)$$

This formula is very impractical as it can only be calculated numerically and we still lack a way to determine σ precisely. Therefore, when estimating the needed window size, 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 8) can be used to approximate the smallest required window size for a given probability p_{accept}.

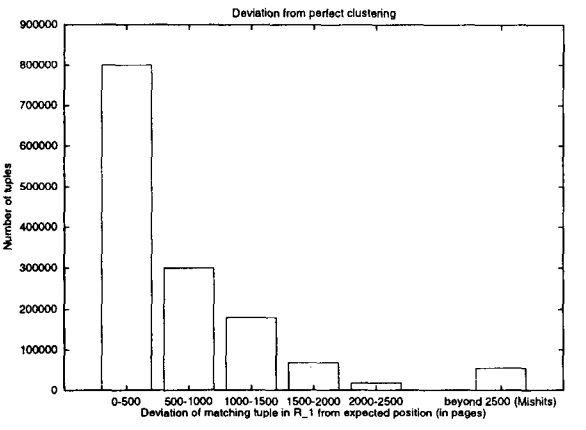

Figure 8: Histograms for measuring deviation from perfect clustering

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 Sun 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 [28]. We joined the relations *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.

Order
orderkey
custkey
orderstatus
totalprice
orderdate
orderpriority
clerk
shippriority
comment

Lineitem
orderkey
partkey
suppkey
linenumber
quantity
extendedprice
discount
tax
returnflag
linestatus
shipdate
commitdate
receiptdate
shipinstruct
shipmode
comment

Table 4: Relations Order and Lineitem from TPC-D

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 [6, 26]. For the index nested-loop join we indexed the attribute *orderkey* on the relation *Order* with a B^+-tree using the Berkeley Database package [2].

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 block-wise nested-loop join, GRACE hash join, and index nested-loop join for different buffer sizes. We did 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 [12, 26]. Table 5 summarizes the parameters for the benchmarks. As can be seen the chosen buffer size is at most $\frac{1}{50}$ of the size of the relations. This is a realistic assumption for Data Warehouses in which huge relations can be found.

Parameter	*Value*
Page Size	4 KByte
Size of *Order*	44,475 pages
Cardinality of *Order*	1,500,000 tuples
Size of *Lineitem*	189,635 pages
Cardinality of *Lineitem*	6,001,215 tuples
Buffer size (window size) for Diag-Join	300 - 4000 pages (1.17 MByte - 15.62 MByte)
Step size (Window size/5)	60 - 800 pages
Buffer size for Nested-loop join (block-wise and index)	300 - 4000 pages (1.17 MByte - 15.62 MByte)
Buffer size for GRACE join	300 - 4000 pages (1.17 MByte - 15.62 MByte)

Table 5: Parameters used for benchmarks

4.2 Benchmark results

4.2.1 Tuning the Diag-Join algorithm

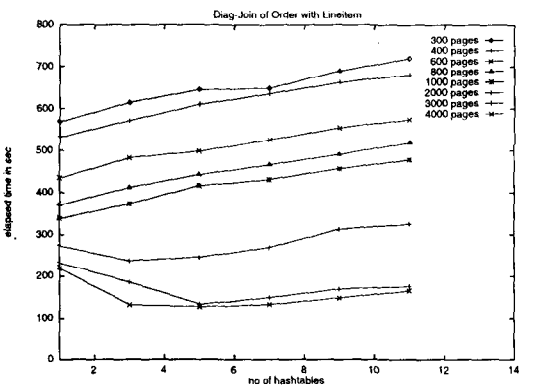

Figure 9: Granularity of hash tables

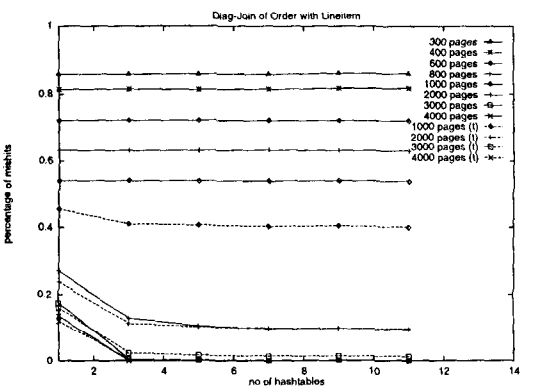

Figure 10: Percentage of mishits

[2] Berkeley DB toolkit: http://www.sleepycat.com

When joining relations with Diag-Join, we have to choose the right step size and buffer size for the window. 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 is irrelevant, 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 in Figure 9. Reducing the step size further does not improve the mishit ratio significantly. The run-time might even deteriorate as it 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. In general this turned out to be a good compromise between optimizing the step size and the search time.

In Figure 10 the percentage of mishits in the relation *Lineitem* is depicted. The results of these measurements are straightforward. The more buffer we allocate, the lower 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. (The curves marked with (t) are theoretical values assuming that the deviation of the relations from perfect clustering can be described by a normal distribution (see Section 3.5).)

4.2.2 Comparison with other join algorithms

In this section we compare Diag-Join with block-wise nested-loop join, GRACE hash join, and index nested-loop join. The results for total runtime of all algorithms for joining the relations *Order* and *Lineitem* on the attribute *orderkey* are shown in Table 6. Block-wise Nested-loop join is used as a reference, not as a serious competitor.

Total costs

Block-wise 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 $||Order||$) Diag-Join easily 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 medium-sized buffers (between 1000 and 3000 pages) Diag-Join is still faster than GRACE hash join and only for very small buffer sizes (<1000 pages or 2% of $||Order||$) GRACE hash join performs better. 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 11). This takes at least some of the load off 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.

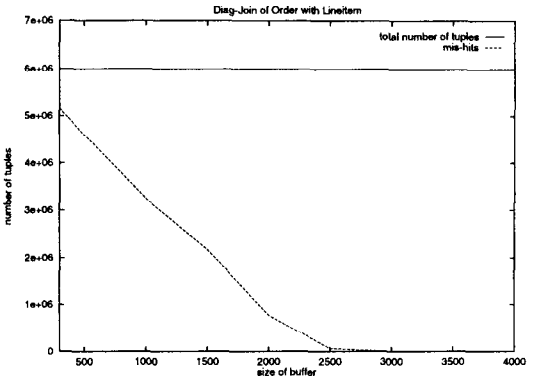

Figure 11: Total number of mishits

Diag-Join also performs much better than index nested-loop join. Although index nested-loop join also profits from the clustering of *Order*, we have to access the tuples indirectly through a B^+-tree, which leads to a much higher overhead than hash table lookups.

CPU-based costs

Let us now have a look at the CPU-based costs of the join algorithms (see Table 7).

The more available memory we have, the lower the costs of the block-wise 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 index nested-loop join are very high. Some of these costs could be reduced by implementing a B^+-tree customized for AODB. However, this will not reduce the costs for searching the inner nodes of the tree, which will always be higher than (maximal) 5 lookups in hash tables.

buffer size (4K pages)	300	400	500	600	800	1000	1500	2000	2500	3000	3500	4000
Algorithm	total elapsed time in sec											
Diag-Join	557	571	441	432	396	333	284	231	194	129	127	126
GRACE hash	552	544	424	424	355	296	300	300	301	296	296	300
Index Nested-loop	-	-	-	-	-	-	-	-	-	-	-	1378
Block-wise Nested-loop	16003	12000	8681	8059	6032	4598	2975	2369	1802	1559	1314	1252

Table 6: Total runtime of join algorithms

buffer size (4K pages)	300	400	500	600	800	1000	1500	2000	2500	3000	3500	4000
Algorithm	elapsed CPU time in sec											
Diag-Join	286	288	271	273	257	215	177	146	123	86	74	72
GRACE hash	194	202	192	205	194	185	186	187	188	191	188	185
Index Nested-loop	-	-	-	-	-	-	-	-	-	-	-	1265
Block-wise Nested-loop	7701	5939	4737	4093	3120	2542	1691	1327	1041	890	758	703

Table 7: CPU-based costs of join algorithms

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 Table 8). 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.

I/O-based costs

The I/O-based costs are displayed in Table 9. For the block-wise 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 becomes 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.

Index nested-loop join also buffers pages of *Order* in main memory. When loading theses pages into memory, however, they are not necessarily accessed sequentially. Therefore seek and latency time is considerably higher for index nested-loop join than for the other join algorithms.

When allocating large buffers for Diag-Join ($\geq$ 3000 pages, which corresponds to about 6% of the size of *Order*), all we have to do is to read *Order* and *Lineitem* once and we are finished. Hence we have low 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 considerably faster than block-wise/index nested-loop join). Diag-Join needs sufficient memory (about 6% of $||R_1||$ 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 $\frac{\text{buffer size}}{R_1} \cdot R_N$ of the tuples in R_N will find a matching tuple in the first phase.

5 Conclusion and future work

We developed a join algorithm, called Diag-Join, for any environment in which joining relations (or extents in object-oriented DBMS) clustered by time of creation is not unusual. We take advantage of the fact that new incoming data is appended at the end of relations (or extents), resulting in a clustering of the tuples (or objects) by time of creation. When this is the case,

buffer size (4K pages)	300	400	500	600	800	1000	1500	2000	2500	3000	3500	4000
Algorithm	elapsed CPU time in sec											
Diag-Join (1st phase)	60	60	61	63	64	66	69	71	73	75	76	76

Table 8: CPU-based costs for the first phase of Diag-Join

buffer size (4K pages)	300	400	500	600	800	1000	1500	2000	2500	3000	3500	4000
Algorithm	elapsed I/O time in sec											
Diag-Join	270	267	169	158	139	118	103	85	65	43	43	43
GRACE hash	357	341	232	219	160	111	114	113	113	104	108	115
Index Nested-loop	-	-	-	-	-	-	-	-	-	-	-	113
Block-wise Nested-loop	8302	6061	3944	3965	2912	2056	1284	1042	761	668	556	548

Table 9: I/O-based costs of join algorithms

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 block-wise nested-loop join, GRACE hash join, and index nested-loop join revealed the impressive performance of our join algorithm. It ran two and a half times faster than GRACE hash join (the latter being on equal grounds with hybrid hash join in our case) and considerably faster than block-wise/index nested-loop join.

Diag-Join can be improved further by integrating it tightly into the join algorithm executed in the second phase. For example, the merging phase of Diag-Join can be coupled with the partition phase of GRACE hash join, i.e. all tuples that do not match are immediately partitioned. This would avoid the first scanning step of GRACE hash 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.

Our next goal is to derive accurate (and not overly complex) methods for estimating the costs of a Diag-Join operator in a query-plan beforehand. This includes finding a measure for the degree of "clusteredness" of relations and the measurement of the effect of various other relational operators on the "clusteredness".

Acknowledgments

We would like to thank Beate Rossi and Wolfgang Scheufele for carefully reading a first draft of this paper. We also thank the anonymous referees for their useful comments.

References

[1] M. W. Blasgen and K. P. Eswaran. On the evaluation of queries in a relational database system. Technical Report IBM Research Report RJ1745, IBM, 1976.

[2] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proc. of the 10th VLDB Conference*, pages 323–333, Singapore, August 1984.

[3] D. DeWitt and R. Gerber. Multiprocessor hash-based join algorithms. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 151–164, Stockholm, Sweden, 1985.

[4] D. DeWitt, R. Katz, F. Ohlken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 1–8, 1984.

[5] D. DeWitt, J. Naughton, and D. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Int. Conf. on Parallel and Distributed Information Systems*, Miami Beach, Fl, 1991.

[6] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the systems software of a parallel relational database machine: GRACE. In *Proc. Int.*

Conf. on Very Large Data Bases (VLDB), pages 209–219, 1986.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, Massachusetts, 1995.

[8] G. Graefe. Sort-merge-join: An idea whose time has(h) passed? In *Proc. IEEE Conference on Data Engineering*, pages 406–417, Houston, TX, 1994.

[9] G. Graefe, A. Linville, and L. Shapiro. Sort versus hash revisited. *IEEE Trans. on Data and Knowledge Eng.*, 6(6):934–944, Dec. 1994.

[10] L.M. Haas, M.J. Carey, M. Livny, and A. Shukla. Seeking the truth about ad hoc join costs. *VLDB Journal*, 6(3):241–256, 1997.

[11] T. Härder. Implementing a generalized access path structure for a relational database system. *ACM Trans. on Database Systems*, 3(3):285–298, 1978.

[12] E.P. Harris and K. Ramamohanarao. Join algorithm costs revisited. *VLDB Journal*, 5(1):64–84, 1996.

[13] W. H. Inmon. *Building the Data Warehouse (2nd ed.).* John Wiley & Sons, 1996.

[14] M. Kamath and K. Ramamritham. Bucket skip merge join: A scalable algorithm for join processing in very large databases using indexes. Technical Report CS-TR-96-20, University of Massachusetts, 1996.

[15] C. Kilger and G. Moerkotte. Indexing multiple sets. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 180–191, Santiago, Chile, Sept. 1994.

[16] W. Kim, K. C. Kim, and A. Dale. Indexing techniques for object-oriented databases. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 371–394, Massachusetts, 1989. Addison Wesley.

[17] R. Kimball. *The Data Warehouse Toolkit.* Jon Wiley and Sons, Inc., New York, 1996.

[18] M. Kitsuregawa, M. Nakayama, and M. Takagi. The effect of bucket size tuning in the dynamic hybrid GRACE hash join method. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 257–266, 1989.

[19] R. Lorie and H. Young. A low communication sort algorithm for a parallel database machine. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 135–144, 1989. also published as: IBM TR RJ 6669, Feb. 1989.

[20] J. Menon. A study of sort algorithms for multiprocessor DB machines. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 197–206, Kyoto, 1986.

[21] P. Mishra and H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

[22] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned join method using dynamic destaging strategy. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 468–478, 1988.

[23] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, Sep 1995.

[24] B. Salzberg, A. Tsukerman, J. Gray, M. Stewart, S. Uren, and B. Vaughan. FastSort: an distributed single-input single-output external sort. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 94–101, 1990.

[25] 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.

[26] L.D. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3):239–264, September 1986.

[27] D. Shin and A. Meltzer. A new join algorithm. *SIGMOD Record*, 23(4):13–18, Dec. 1994.

[28] Transaction Processing Council (TPC). TPC Benchmark D. http://www.tpc.org, 1995.

[29] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2), 1987.

[30] Z. Xie and J. Han. Join index hierarchies for supporting efficient navigation in object-oriented databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 522–533, 1994.