

An Efficient Hybrid Join Algorithm: A DB2 Prototype

Josephine Cheng, Don Haderle, Richard Hedges*, Bala R. Iyer, Ted Messinger#, C. Mohan#, Yun Wang

Data Base Technology Institute, IBM Santa Teresa Laboratory, San Jose, CA 95161, USA

*IBM, 895 Don Mills Road, 1 Park Center, North York, Ontario M3C 1W3, Canada

#Data Base Technology Institute, IBM Almaden Research Center, San Jose, CA 95120, USA
bala@ibm.com, mohan@ibm.com

Abstract Many commercial relational data base systems provide two join methods: 1) Nested Loop join and 2) Sort Merge join. Nested Loop join exploits indexes on the inner table's join column. Sort Merge join benefits from the efficiency of bulk sequential disk accesses. Both provide good performance when selected correctly by the query optimizer, but could prove prohibitively expensive if selected incorrectly. We describe a new method called *Hybrid join*. We first sort the outer table on the join column. Then, the outer is joined with the index on the join column of the inner. The inner tuple is represented by its surrogate, equivalent of its physical disk address, which is carried in the index. The partial join result is sorted on the surrogate and then the inner table is accessed sequentially to complete the join result. Local predicate filtering can also be applied before the access of the inner relation through index AND/ORing. We also present efficient methods for skip sequential access and prefetching of logically discontinuous leaf pages of B⁺-tree indexes.

1. Introduction

Because of the common practice of normalization which separates data into many tables to reduce redundancy, and because of the growing usage of recursive queries, the join operator is very often used while querying a data base. Join queries over large tables execute for a long time and choice of the right join method can reduce the elapsed time often by orders of magnitude. Many studies of join methods have been reported in the literature [BIEs76, DKOSS84, Kim80, Vald87]. Many commercial systems support the Nested Loop and/or the Sort Merge join methods, based on the early System R research which concluded that the Nested Loop and the Sort Merge join methods were always optimal or near optimal over the range of parameter space explored in [BIEs76]. [BIEs76] did not consider the possibility of using bulk I/Os (i.e., multiple pages being read or written via a single I/O call) and prefetching of pages to reduce I/O costs and to better balance I/O and CPU processing. When data was accessed via indexes, in order to provide clustered access to the data, the idea of sorting the row or record identifiers of the selected keys from the indexes before performing data page accesses was not considered. Our paper is not intended to compare all join methods as was done in [BIEs76], but to describe a new join method using the indexes on join columns that often

exist on large relations. We combine the good features of the two established join methods.

In the Nested Loop join method, for each row in the outer table that satisfies the local predicates, we look for the matching rows in the inner table (usually, via an index). This method makes efficient use of an index on the join column of the inner table. It is most often the best method when the join column values passed to the inner table are in sequence and the join column index of the inner table is *clustered* (i.e., rows of the inner table are approximately in the same physical sequence on disk as the sequence of key values in the join column),¹ or the number of rows retrieved in the inner table via the index is small. However, there are many drawbacks with this method. They are: (a) **Synchronous I/O** on data pages when the outer table presents join column values in other than the clustering sequence of the inner table and on index pages when the outer table's sequence is different from the sequence of the index key, (b) **index probes** which require a search from the root to the leaf for every key presented to the index when the outer table's join column values arrive in a sequence other than the sequence of the key in the inner table's join column index, and (c) **inefficient processing of duplicate** outer table join column values under the same circumstances.

With the Sort Merge join method, once the two tables are ordered on the join column(s), the join can be done similarly to the merge phase of a sort operation. A table can be ordered via sort or accessed via an index to achieve access in join column sequence. Local predicates on a table can be applied before the sort. Sort Merge join can make efficient use of sequential access by prefetching multiple data pages, amortizing disk seek and latency overhead over multiple page transfers. Sort Merge join is most often the best when qualifying rows of inner and outer table are large and the *join* predicate does not offer much filtering. The disadvantage of Sort Merge join is that it cannot use the *join column* index for filtering - rows that do not participate in the join result may also be sorted - consuming extra CPU and I/O resources.

As will be shown by our analysis later, the performance differential between Nested Loop and Sort Merge joins can vary by an order of magnitude. This implies that the cost of incorrectly choosing a join method may be very high. Today's optimizers may choose a join method incorrectly because of insufficient or incorrect statistics, programming language variables in the query (whose

¹ A physical scan of a table (called *table scan*) which is clustered on the join column will result in the table's rows being retrieved *approximately* in the sequence of the join column values. This type of an access is exploited in the Hybrid join method in order to avoid sorting all the qualifying rows on the join column value.

values are not known at query optimization time), or correlated column values across tables.

The rest of the paper is organized as follows. In section 2, we present the Hybrid join algorithm. Modelling and analysis are presented in section 3. Section 4 discusses measurements. A summary of our work is included in section 5.

2. The Hybrid Join Algorithm

2.1. Overview

The Hybrid join method requires that an index exist that includes all the join columns of the inner table or that one or more indexes exist covering all the join columns of the inner table, where index "ANDing" and/or "ORing" (see [MHW90]) may be performed to achieve the result of a single composite index on all the join columns.

Hybrid join makes efficient use of the inner table's join column index. To scan the inner table index efficiently for the rows matching those of the outer table, the join column values of the outer table should be passed in sequence or, at least, approximately in sequence. This can be achieved by (1) sort, (2) index access on the join column of the outer table, or (3) even a table scan when the outer table is clustered on the join columns. The index of the inner table is scanned to retrieve the row identifiers (RIDs) of the inner table's rows that matched the join column value of the outer table. An efficient index scan algorithm is described in the section "2.3. Skip Sequential Index Scan". Efficiency is achieved by remembering the position on the leaf page of the previously found index key.

Rows in the outer table with duplicate join column values can be easily detected since they are more or less in join column sequence. The inner's index does not have to be rescanned for duplicates. Hybrid join can make efficient use of a method that we call *skip sequential prefetch*. It is a mechanism to prefetch several sequential, but not necessarily contiguous, pages. A list of sorted RIDs is passed to the buffer manager to prefetch the data pages containing the rows identified by the RIDs. CPU and I/O processing may be overlapped. A qualifying data page of the inner table is fetched only once to retrieve all the rows joining with the outer table. Hybrid join can make efficient use of index ANDing and/or ORing, as described in [MHW90]. In Sort Merge join, only the index on the columns of the local predicates is exploited. In Hybrid join, indexes on the columns of both the local predicates and the join predicates are exploited. In ANDing and/or ORing, we merge the sorted RID list obtained from the index on the join column with the RID list obtained from indexes on other inner table columns involved in the inner table's local predicates.

When there are *clustered* indexes on the join columns of both the outer and the inner tables, the Hybrid join method still provides efficient index scan, efficient duplicate processing and skip sequential access to the inner table. In this case, the join column values of the outer table and the RIDs of the inner table are already in approximate order. Therefore, there is no need to sort or materialize the rows that qualify from the outer table. One may take m rows of the outer table that satisfy the

local predicate, where m is the memory size available for internal workfile (temporary table) management.

The Hybrid join method may be applied to an n -table join by recursively applying the Hybrid join algorithm ($n-1$) times. Hybrid join may also be combined with the other join methods in an n -table join, where $n > 2$.

2.2. Main Algorithm

Hybrid join consists of the following steps:

STAGE 1

Step a Take n rows of the outer table that satisfy the local predicates, where n is equal to

m for highly clustered indexes on the join column of both the inner and the outer tables (with the outer table being accessed via a clustered index or a table scan), where m is determined by the memory size available for join processing

x for all other cases, where x is the total number of rows in the outer table that satisfy the local predicates

Step b The n retrieved outer table rows should be in join column sequence so that the inner table index needs to be scanned only once for each key value. The sequencing may be achieved via:

- sort,
- access via the join column index of the outer table,
- table scan when a highly clustered index exists on the join column(s) (in this case the rows are only in approximate sequence which suffices), or
- only one row qualifying from the outer table

Step c Access the index of the join column of the inner table using the join column values of the n rows to retrieve the matching inner table's rows' RIDs. The RIDs can be retrieved by an efficient index scan as described in the section "2.3. Skip Sequential Index Scan".

Step d No composite rows will be formed when the number of qualifying rows in the outer is one ($x=1$). If all the columns being selected can be fetched from the inner table's index, then the join operation is completed. Rows with selected columns of the outer table and the selected columns of the inner table from the index can be returned. Otherwise, we form the composite row with the selected columns of the outer table and the RID of the inner table either in memory (when $n=m$) or in a workfile (or temporary table).

Step e For any duplicate key value in the outer table, the matching RID list is replicated without looking up the index tree.

STAGE 2

Step a If the inner table's join column index is a clustered index and there is no index ANDing for local predicates, then the RIDs should already be in approximate order and sorting the RIDs is unnecessary. Otherwise, we sort these composite rows on the inner table's RID value.

Step b Index ANDing and/or ORing will be applied if there are indexes on the columns of the local predicates of the inner table.

STAGE 3

The RID list of the inner table is passed to the buffer manager to enable skip sequential prefetch of the inner

table's pages. The RID list will not contain any duplicates so that positioning is done only once on each row. Rows that satisfy the local predicates and the join predicate will be returned.

2.3. Skip Sequential Index Scan

In this section, a method for efficiently scanning an index for different input key values is presented. The type of index that we consider in this paper is the B^+ -tree [Moha90] in which the actual key values with associated row identifiers are stored only in the leaf pages. The motivation is to reduce the amount of time and the number of page accesses it takes for the index manager to move forward (or backward) to a leaf page with a specific higher (lower) key value given that a position is currently held on a preceding (succeeding) page with a particular lower (higher) key value. Such a capability would be very useful in the Hybrid join method when moving around in the inner table's join column index in response to the input keys from the outer table's rows.

In addition to the usual contents of the leaf and nonleaf pages in a B^+ -tree (see, e.g., [Moha90] for a description of those), the index manager also maintains a parent pointer in all the leaf and nonleaf pages. This pointer is maintained as accurately as possible without incurring too much overhead. This is accomplished by making actions which do probes (for key inserts, deletes or fetches) ensure that each child (leaf or nonleaf) page that is accessed contains the correct parent page pointer. If the parent pointer has a wrong value, then the correct value is put in. The probe action will not necessarily log this update or let the buffer manager know that the child page has been updated as a result. The costly operation that will be avoided at the time of a nonleaf page split is the updating of the parent pointer of all those child pages that get a new parent. Subsequent probe actions that visit those children will be made to fix up the wrong pointers.

Given that the parent pointer is maintained in every page, when we are positioned at a leaf page with a certain key value, when another key value is provided to position on, we compare the new key to the smallest and the highest key values in the current page. If the new key is not in that range, then we access the parent and check if any other child of that parent could contain the new key. If such a child is identified, then we access that child. If the parent does not cover the range that includes the new key, then we climb further up the tree and repeat the search. If the new key is in the current leaf page, there is little work to be done. If the new key is equal to the highest (smallest, respectively) key in the current page, then we have to extract the needed information from the current page and also access the next (previous) leaf page by following the next (previous) pointer at the leaf page. This results in a "zig-zag" pattern of accessing the different levels of the tree.

2.4. Skip Sequential Prefetch of Logically Discontiguous Index Leaf Pages

If the previous subsection's technique alone is used, then it may result in a number of synchronous, single page I/Os for the index pages (mostly leaf pages, assum-

ing that most of the nonleaves are always cached in the buffer pool). For the reasons explained before, we would like to do multipage I/Os and prefetch pages before they are needed. In this section, we present a method for prefetching a set of logically discontiguous leaf pages based on a set of key values that are of interest. This technique will enable us to do batched I/Os on the inner table's index to bring in the required leaf pages in parallel with CPU processing, given a set of join column values from the qualifying outer table rows. We are more concerned with prefetching the leaf pages than the nonleaf pages since it is highly likely that the number of leaves examined will be significantly more than the number of nonleaves that are examined. The idea is to sort, if necessary, the list of keys and then access the parents of leaf pages which contain those keys (using the kind of "zig-zag" access described in the last subsection, but restricting the accesses at the lowest level of the tree to be at the parents of the leaf level) and identify the leaf pages which need to be accessed. Once we generate the list of leaf pages to be accessed, then we can initiate prefetches of those logically discontiguous leaf pages. Note that there may be different ranges of logically contiguous leaf pages that need to read in from disk.

While the logic that does the prefetch would use the technique of this subsection to identify the leaf pages of interest, the logic that does the actual join would follow the technique of the last subsection. This is important since between the time of prefetch and the time of the actual access by the join logic, keys may get moved around due to other transactions' activities.

2.5. Hybrid Join Advantages

The benefits over Nested Loop join are:

- Providing clustered access to the inner table even when the index on the join column is unclustered, thereby reducing the number of synchronous single page I/Os.
- Exploiting skip sequential prefetch to reduce the number of synchronous I/Os even when using a clustered index.
- Reducing CPU and, perhaps, I/O overheads because of efficient duplicate processing. The same row which satisfies duplicate join column values of the outer table will not be searched for more than once.
- Reducing CPU and I/O overhead involving index nonleaf pages with an efficient index scan by remembering the position of the leaf page for the last key searched.
- Much less sensitivity to incorrect statistics, as compared to Nested Loop join. Even in regions where Nested Loop join is the best, the cost of Hybrid join is very close.

The benefits over Sort Merge join are:

- Exploitation of the join column index for filtering, thereby reducing the access of unnecessary pages and rows.
- Exploitation of the index ANDing of local predicates with the join column index.
- Much less sensitivity to incorrect statistics, as compared to Sort Merge join.

3. Modelling and Analysis

During the early development of the algorithm, we desired to understand the performance improvements because of Hybrid join. The intent of the performance model is to estimate the elapsed time of an SQL query involving a join, running on a single processor with no load other than that imposed by the join query. The analysis is based on expected execution costs for specific data base manager functions. For a data base manager supporting the functions that we assume, the cost of the join can be found by simply plugging in the execution costs of the different functions. For data base managers that do not support a specific function, one would need to cost the composite functions needed to implement the function desired. In this sense, the model and the analysis are general enough to apply to data base managers other than the one we had in mind during our analysis.

A summary of the cost formulae used in our performance model is given in the following. We focus on a two table join. Let T1 denote the outer table and T2 the inner table. Neither table is clustered on the join column. We will assume that the only possible access path for T1 is a table scan, while T2 could be accessed via a table scan or by an index on its join column. We will describe our cost formulae for Sort Merge join and for Nested Loop join. These costs reflect the cost of the 'best of breed' algorithm for each join method, as opposed to specific algorithms in any specific product.

In Figure 1, we summarize the access paths that we assumed for the three types of joins. For example, for Hybrid join, we assume that the outer table is accessed via a table scan, while the index on the join column is used in an index scan of the inner table. The rows of the inner table, when they are actually accessed, are accessed by using the **Skip Sequential Access Method**. The access method entails collecting all the RIDs of the rows to be accessed (RIDs are essentially disk addresses), sorting the RIDs and accessing the rows in this sorted order. The advantage is that we reduce unnecessary disk arm movement and increase the number of pages read per I/O.

3.1. Hybrid Join

We review below the 3 phases of Hybrid join.

SEL.PRO.SRT: Selection, projection and sort of T1
 IDXJOIN: Merge T1' runs, scan inner index and form (T1",T2RID)

FRMRSLT: Access T2 rows (skip seq. access), apply local predicate, return join result to user

In the first phase, which we denote symbolically as the SEL.PRO.SRT phase, the outer table T1 is accessed, local

predicates are applied, required columns are projected, and the selected rows are introduced into a sort tournament tree. Whenever we refer to the rows of table T1 after selection and projection, we will refer to them as rows of T1'. Sorted runs are formed in this phase using the tournament sort method, and they are written to workfiles. Throughout this analysis, we assume that the sorted runs may be merged in one pass. In the second phase, denoted as the IDXJOIN phase, the sorted runs of T1' are merged. While the runs are merged using a tournament tree, a scan of the join column index of T2 proceeds, by looking for matches in the inner table, for every T1' row in the merged stream. Whenever a match is found, an intermediate result (T1",T2RID) is formed and inserted into another tournament tree, which sorts based on the T2RID. Sorted runs so produced are written into workfiles. We refer to rows of T1' that find a match in table T2 as rows of T1". In the final phase, FRMRSLT, these sorted runs are accessed from workfiles and merged. The rows (T1",T2RID) are output in T2RID order by the merge phase. We model the skip sequential access method that retrieves the matching rows of T2. The access occurs during the merge. Whenever a T2 row is accessed, local predicates are applied. For qualifying T2 rows, the join results are formed, and the resultant rows are returned to the user.

Below are the cost formulae used to evaluate the CPU, I/O and elapsed times for Hybrid join. We have provided, in the section "Appendix A. Key to the Notation", a key to interpret the meanings of the different variables used in all the following formulae.

SEL.PRO.SRT

$$\begin{aligned} \text{CPU Time} = & \text{NRows} * (\text{DM} + \text{FF} * (\text{RDS} + \text{EXTKEY} + \text{TTSRT} + \text{WKINST})) + \\ & \text{DataPgs/SqPFsz} * \text{PFIOPL} + \text{WkfPgs/WIOSz} * \text{WIOPL} + \\ & (\text{DataPgs} + \text{WkfPgs}) * \text{PgMgmt} + \text{DataPgs} * \text{LKUnlk} \end{aligned}$$

$$\text{I/O Time} = \max(\text{DataPgs/SqPFsz} * \text{SqPFIOt}, \text{WkfPgs/WIOSz/WkfDsk} * \text{WIOTime})$$

$$\text{Elapsed Time} = \max(\text{CPU Time}, \text{I/O Time})$$

The CPU time for the first phase is modelled by the average cost (DM) of applying predicates to each of the NRows rows. For every row that qualifies, and there are FF*NRows of these, the CPU cost consists of the cost (RDS) of moving the row across to RDS (the query processing component), extracting the sort key (EXTKEY), the cost of the tournament tree sort (TTSRT), and the cost of inserting the row into the workfile (WKINST). Besides these, there are I/O initiation costs: (1) to initiate read I/Os on the table being scanned (PFIOPL) and (2) for writing the workfile pages (WIOPL). These types of I/Os are assumed to read or write multiple pages at a

	Nested Loop		Sort/Merge		Hybrid	
	Index	Data	Index	Data	Index	Data
Outer Table	N/A	Table Scan	N/A	Table Scan	N/A	Table Scan
Inner Table	Probe	Via Index	N/A	Table Scan	Scan	Skip Sequential Prefetch

Figure 1: Access Paths Assumed for Different Join Methods

time. Also, there is a cost to pin and unpin in the buffer pool every page of the table being scanned and the workfile pages (PgMgmt). Finally, we assume page level locking and add in the cost of locking and unlocking every page of the scanned table.

The I/O time is estimated next. The number of pages to be fetched from the relation is given by DataPgs. These pages are prefetched in blocks of SqPFSz pages. Each block fetch takes SqPFIOt time. *Concurrently*, sorted runs formed using the tournament tree are written to workfiles on system disks. We assume striping of the workfiles on the system disks, which are WkfDsk in number. Writes are batched, where each batch consists of WIOSz pages. Hence, the write I/O time is $((WkfPgs/WIOSz)/WkfDsk) * WIOTime$, where WIOTime is the average time for writing WIOSz pages to one disk. Note that we account for the fact that reads and writes occur concurrently. We assume that the runs are formed on disks different from the disks that contain the relations joined. Elapsed time is simply the greater of the CPU and I/O times.

The cost formulae used in Phase 2 are listed next. We have not included a detailed explanation of each term. Instead, we have selected mnemonic variable names to help the interested reader understand the meaning of each term.

IDXJOIN

CPU Time =
 $NRowsT1 * FFT1 * (WKFFTC + TTMRG + EXTREC) +$
 $NRowsT1 * FFT1 / Avg.Dup.T1 * (COMPHK + SLIDE +$
 $RETIRID + (Avg.Dup.T2 - 1) * RETNRID) + NLFpgsT2Ix *$
 $COMPHK + NRowsT1 * FFT1 * FFJ12 * Avg.Dup.T2 *$
 $(EXTKEY + TTSRT + WKFINST) + (NpgsWkfT1' +$
 $NLFpgsT2Ix + NpgsWkfT1" * T2RID) * PgMgmt +$
 $NLFpgsT2Ix * LkUnlk + NpgsWkfT1' / WkfPFSz * WkfPFIOPL$
 $+ NLFpgsT2Ix / PFIOSz * PFIOP +$
 $NpgsWkfT1" * T2RID / WIOSz * WIOPL$

I/O Time =
 $\max(NpgsWkfT1' / WkfPFSz / WkfDsk * WPFIOTime +$
 $NpgsWkfT1" * T2RID / WIOSz / WkfDsk * WIOTime,$
 $NLFpgsT2Ix / PFIOSz * PFIOTime)$

Elapsed Time = $\max(\text{CPU Time}, \text{I/O Time})$

We use the following cost formulae for the third phase of Hybrid join.

FRMRSLT

CPU Time =
 $NRowsT1 * FFT1 * FFJ12 * Avg.Dup.T2 *$
 $(WKFFTC + TTMRG + EXTREC) + NRowsT1 * FFT1 *$
 $FFJ12 * Avg.Dup.T2 / Avg.Dup.T1 * (DM + FFT2 * RDS) +$
 $NRowsT1 * FFT1 * FFJ12 * Avf.Dup.T2 * FFT2 * RetUsr$
 $+ (NpgsWkfT1" * T2RID + NpgsT2acc) * PgMgmt +$
 $NpgsT2acc * LkUnlk + NpgsWkfT1" * T2RID / WkfPFSz *$
 $WkfPFIOPL + NpgsT2acc / SSPFIOSz * SSPFIOP$

I/O Time = $\max(NpgsWkfT1" * T2RID / WkfPFSz / WkfDsk *$
 $WPFIOTime + NpgsT2acc / SSPFIOSz * SSIOTime)$

Elapsed Time = $\max(\text{CPU Time}, \text{I/O Time})$

The total time for Hybrid join is the sum of the times for the three component phases.

3.2. Sort Merge Join

Sort Merge join has three phases too, as follows.

SEL.PRO.SRT: Selection, projection and sort of T1
 SEL.PRO.SRT: Selection, projection and sort of T2
 MRGJOIN: Merge T1' runs, merge T2' runs,
 join during merge, return results to user

In the first phase (similar to Hybrid join), which we denote symbolically as the SEL.PRO.SRT phase, the outer table T1 is accessed, the local predicates are applied, the required columns are projected out, and the selected rows are sorted using a tournament tree. The sorted runs formed in this phase are written to workfiles. In the second phase, the same work is done on the inner table T2. The cost formulae used for these two phases are given earlier under the discussion for Hybrid join cost formulae. In the final phase, MRGJOIN, the two sets of sorted runs are merged concurrently. While they are being merged, the merge outputs are joined and the results are returned to the user. The cost formulae used for this phase are given next.

MRGJOIN

CPU Time =
 $NRowsT1 * FFT1 * (WKFFTC + TTMRG + EXTREC) +$
 $NRowsT2 * FFT2 * (WKFFTC + TTMRG + EXTREC) +$
 $NRowsT1 * FFT1 * FFJ12 * Avg.Dup.T2' * RetUsr +$
 $(NpgsWkfT1' + NpgsWkfT2') * PgMgmt +$
 $(NpgsWkfT1' + NpgsWkfT2') / WkfPFSz * WkfPFIOPL$

I/O Time =
 $\max(NpgsWkfT1' + NpgsWkfT2') / WkfDsk * WPFIOTime$

Elapsed Time = $\max(\text{CPU Time}, \text{I/O Time})$

As in the case of Hybrid join, the total time for Sort Merge join, is the sum of the times for the three component phases.

3.3. Nested Loop Join

In contrast to the other join methods, in our model, Nested Loop join has only one phase. The outer table T1 is accessed via a table scan. For every row in the outer table that satisfies the local predicate, we use the join column index on the inner to find out if there is a match. If we find a match, then we access the T2 data page immediately. We model the evaluation of the local predicate against the row accessed. For rows that qualify, we model the cost to form the join result and return it to the user. Cost formulae used are listed next.

CPU Time =
 $NRowsT1 * (DM + FFT1 * (RDS + NixLvlis *$
 $(SLIDE + LkUnlk) + SYNCIOPL + FFJ12 *$
 $(Ret1stRID + (Avg.Dup.T2 - 1) * RetnRID +$
 $Avg.Dup.T2 * (SYNCIOPL + PgMgmt + LkUnlk + DM +$
 $FFT2 (RDS + RetUsr)))))) + T1DataPgs * LkUnlk +$
 $T1DataPgs / SqPFSz * PFIOP + NonLfIxPgsT2 * SYNCIOPL$

SYNCIO Time = $(NRowsT1 * FFT1 * (1 +$
 $FFJ12 * Avg.Dup.T2) + NonLfIxPgsT2) * SYNCIO$

$PFIOTime = T1DataPgs/SqPFsz * PFIOT$

$Elapsed\ Time = \max((CPU\ Time + SYNCIO\ Time), PFIOTime)$

3.4. Modelling Results

Using CPU costs experienced by IBM's DB2 on a model 3090 processor, and I/O costs on model 3380 disks, we explored a large portion of the design space and undertook sensitivity studies. Results from a few studies, illustrative of our methodology, are presented in Figure 2, Figure 3, Figure 4, Figure 5, Figure 6, Figure 7 and Figure 8. In Figure 2 and Figure 3, we analyzed a join of two tables. The outer table has 100K rows, the inner has 1M rows. The outer has 4 columns. Each row occupies 34 bytes. The inner has 4 columns. Each row occupies 40 bytes. The join column values are drawn from a domain of 100K unique values. The outer has no duplicates in the join column, while the inner has an average of 10 duplicates per value in the join column (also, for every row of the outer table).

We study the sensitivity of the elapsed time and the CPU time to the filter factor of the outer table. First, we assume that there is no predicate on the inner table. The cost of a table scan for the outer table is common to all the three join methods. At low selectivities for the outer table, Nested Loop join is the best since it accesses only very few rows of the inner table (i.e., only those that participate in the join). Sort Merge join is the worst since it accesses all the rows of the inner table and sorts them. On the other hand, when the local predicate on the outer table is not selective, almost all the rows from the inner and outer will qualify. Under these conditions, Sort Merge join is the best. In between, over a large range of filter factors, Hybrid join shows the smallest elapsed time. This is because the selectivity of the outer table causes only a corresponding fraction of the inner's rows to be accessed. In our model, Hybrid join pays the cost of an index scan up-front. That is why, it is worse than Nested Loop join for very small selectivities. If we had modelled index probes instead of the index scan, the performance of Nested Loop and Hybrid join would be much closer.

We plot projected CPU times in Figure 3. Note that Hybrid join is the least expensive method of the three over almost the entire range of values for the filter factor of the local predicate on the outer table. For Nested Loop and Hybrid join, there is a near linear increase in cost as the number of rows qualifying from the outer increases. If the number of qualifying rows is small for the outer table, then Nested Loop join has a very small CPU cost, as does Hybrid join. This is because we access the inner table only for those rows that qualify from the outer. When a larger number of rows qualify from the outer, the CPU cost of Nested Loop rises more steeply than the CPU cost for Hybrid join. This is because, in the Nested Loop join model, for every qualifying row of the outer, (1) a search occurs through each level of index, (2) at least one synchronous I/O is initiated for the leaf, and (3) for every match, a synchronous I/O is initiated for the data page. Hybrid join optimizes these costs, by avoiding synchronous I/Os against the index and data and by reducing accesses to the index and data pages by ordering the accesses by key values and RIDs. Over

a vast range of parameters, Hybrid join is superior to both Nested Loop and Sort Merge join, in terms of CPU time as well as elapsed time.

In Figure 4, we see the effect of applying a predicate to the inner table. 50% of the rows from the inner table are assumed to qualify. Sort Merge join benefits the most from the selectivity of the predicate. The number of rows to be sorted and merged in the inner (the bigger table) is reduced by 50%. The cost of Nested Loop join is not impacted much because the number of accesses to the inner tables' index or data pages is not changed, even though the size of the join result is reduced. The same argument applies to our model of Hybrid join. This explains why the crossover between Hybrid join and Sort Merge join, in the elapsed and CPU time curves in Figure 4 and Figure 5, respectively, occur at a lower value of filtering for the outer local predicate, as compared to the crossovers in Figure 2 and Figure 3. Hybrid join would need to use index ANDing and/or ORing [MHWC90] to derive the benefit of the inner local predicate. We have not modelled index ANDing and/or ORing.

Next, we explore the selectivity of the join predicate. We increase the size of the set from which join column values will be picked in our analysis. All 100K outer rows are assumed to have unique values and all 1 M inner rows are assumed to have unique values out of a total of 10M possible values. For any outer row, we assume that the probability of finding a match in the inner is 0.1. The elapsed times and CPU times for the three algorithms are given in Figure 6 and Figure 7, respectively. Note how join predicate filtering makes Hybrid join the method of choice.

We also evaluated the amount of time spent in I/O in Figure 8. Note that Nested Loop join is primarily I/O intensive. For example, when the local predicate filter factor of the outer table is 1, the I/O time is 2471.7 seconds out of an elapsed time of 2580.6 seconds. The synchronous I/O time we have used is 22.54 msec. About 110,000 synchronous I/Os could be executed in 2471.7 seconds. In contrast, very little time is spent in I/O for the Sort Merge and Hybrid joins.

4. DB2 Measurements

A prototype implementation of the Hybrid join was done with the DB2 V2.2 code using the existing constructs of both the Nested Loop join and the Sort Merge join in a couple of person-months. The efficient index scan described in this paper was not implemented. Therefore, the prototype is expected to underperform a full-fledged implementation. The prototype was installed on an IBM 3090-300 mainframe computer running IBM's MVS/ESA 3.1 operating system. Although the machine has 3 processors, for all practical purposes, only one processor was used during the measurements. The machine had 256 MB of byte addressable main memory and 512 MB of page addressable expanded memory. However, the DB2 data base buffer was limited to 1000 4 KB pages (i.e., 4 MB in total). Any extra pages could not reside in the buffer. The reason to limit the data base buffer to a size smaller than memory was to reflect the buffer availability to our join, if it were to run in a multiuser environment. The data base buffer was partitioned into two equal parts. One buffered permanent data base pages

and index pages. The other buffered workfile pages created to hold sort runs.

We used three of IBM's 3380 disks - one for each table joined and one for the index. The disks were connected to different IBM 3880 disk control units. For table sizes, we picked the same parameter values as in the model. T1, the outer table had 100,000 rows: 4 columns, 34 bytes per row. We initialized column 3 with independently generated random numbers in the range 0 - 99. Column 4 was initialized with independent random numbers in the range 0 - 9999999. T2, the inner table had 1,000,000 rows: 4 columns, 40 bytes per row. Columns 3 and 4 of T2 were initialized in the same way as in T1. In addition, an index was defined on column 4 of T2. The schema of the two tables were defined as follows:

T1

C1 (CHAR (15))	C2 (CHAR (8))	C3 (INTEGER)	C4 (CHAR (7))
----------------	---------------	--------------	---------------

T2

C1 (CHAR (21))	C2 (CHAR (8))	C3 (INTEGER)	C4 (CHAR (7))
----------------	---------------	--------------	---------------

No columns were allowed to have NULL values

Finally, the join query is specified in SQL as follows:

```
SELECT T1.C1, T2.C1
FROM T1, T2
WHERE T1.C4 = T2.C4 AND T1.C3 < nn;
```

As in the model, the outer table had 100K rows. Values in its join column were drawn randomly from a domain of 10 million unique values. The inner table had 1 million rows and the join column values were randomly selected from the same 10 million unique values. As a result, duplicates of the same value in the join column of the same relation were rare. Every row of the outer has a 1 in 10 odds of finding a match in the inner. This is close to the case modelled earlier whose results are given in Figure 6, Figure 7 and Figure 8. The measurements results are shown in Figure 9, Figure 10 and Figure 11. Differences between the prediction of the analytical model and the prototype measurements are explained next.

First, let us compare the plot of 'elapsed time' against 'local predicate filter factor for the outer table', i.e. Figure 6 with Figure 9. Note first that the elapsed time for Nested Loop join rises more steeply in the model prediction as compared to the measurement. There are two reasons. To understand them, consider the case when the outer table's local predicate filter factor value is 1. As explained earlier, the model predicted 110,000 I/Os. The measurement shows 100,000 synchronous I/Os. The small decrease is because of reuse of some of the data pages from the main memory data base buffer. In the model, we assumed that every data page access for Nested Loop join entailed an I/O. Only for the top two levels of the index pages did we assume hits in the main memory buffer. Next, the prototype measurements were made in a single user environment where there was no contention in the I/O subsystem and I/O delays were around 16 msec or less. In the model, we used an average I/O delay of 22.454 msec, an overestimate. The

two differences account for the difference in the slope of the Nested Loop join elapsed time curves, since the elapsed time of Nested Loop join consists of mainly synchronous I/Os.

The second noteworthy difference is that the elapsed time of the measured Sort Merge join is higher than what was predicted. There are two reasons for this. (1) We modelled the algorithm where the join occurred concurrently with the final merge of the sorted runs of the two relations being joined. But, in the prototype, the runs from the two relations were merged separately, each to one single long run and inserted into a temporary relation. The two relations were then joined. (2) We modelled no synchronous I/Os for Sort Merge join and Nested Loop join. The measurement show a small but significant number of synchronous I/Os for the two join methods in the prototype. This is because the prototype's sequential prefetch mechanism is triggered off only after some number of synchronous I/Os had taken place against a table. Finally, the measured elapsed time for Hybrid join turned out to be larger than the predicted value. We attribute this difference to the excessive CPU cycles consumed by the Hybrid join prototype. This can be observed by comparing Figure 7 and Figure 10. In the prototype, the index on the join column of the inner table is probed from root to leaf for every qualifying row of the outer table. The model assumed that the more efficient index scan is used instead. This is the reason the CPU cost of Hybrid join is measured to be more than the CPU cost of Nested Loop join, while we predicted the reverse.

5. Summary

In this paper, we have proposed a new join method, called Hybrid join, which uses the join index filtering and the skip sequential prefetch mechanism for efficient data access. We also showed an efficient index scan using information from the previous index access. The Hybrid join algorithm was shown to be a good algorithm via modelling and measurements. We also showed that the model's predictions are reasonably close to the actual measurements.

We did not compare Hybrid join with Hash join due to the following reasons. If we use an N-way merge, where N is large (e.g., 256), then merge will often be completed in a single pass.² As a result, Sort Merge join may be completed in two passes over the rows to be joined. We believe that Hash join exhibits a similar behavior, assuming that neither table fits in memory. The first pass is used to divide qualifying rows into hash buckets and the second to join rows in the same bucket. Modulo the CPU cost differences between sorting a row versus hashing a row, the other costs for the two join methods are comparable. We may argue about fast ways to hash a row, or sort a row using offset value coding, but the fact remains that the behavior of these costs with query characteristics do not differ. We were first made aware of these similarities by our colleague Hamid Pirahesh (unpublished work). In the future, we would like to validate

² If the DBMS were to use for sorting a tournament tree with 16K leaf nodes, then on the average we would expect each sort run to have 32K rows. With a 256-way merge, we should be able to sort and merge about 8M rows in one merge pass.

these claims. Also, the other reason for our not considering Hash join is that, typically, Hash join is superior only if no index exists on the join column(s) of the inner table. On the other hand, the Hybrid join method comes into play only if there is such an index.

Two common techniques are used for parallel execution: data partitioning - chopping up the data, and pipelining -- overlapping operations [PMCLS90]. Hybrid join algorithm is easily extended to both techniques as discussed in [CHHIM90]. Other variations and extensions of the algorithm are also described in [CHHIM90].

Acknowledgements We would like to thank Akira Shibamiya for performance consultations, Hong Tie and James Guo for prototyping, Gene Kupersmidt for the 3090 measurements and Pat Selinger for her comments.

6. References

- BIEs76** Blasgen, M., Eswaran, K. *On the Evaluation of Queries in a Relational Data Base System*, **IBM Research Report RJ1745**, IBM San Jose Research Laboratory, April 1976.
- CHHIM90** Cheng, J., Haderle, D., Hedges, R., Iyer, B., Messenger, T., Mohan, C., Wang, Y. *An Efficient Hybrid Join Algorithm: a DB2 Prototype*, **IBM Research Report RJ7884**, IBM Almaden Research Center, December 1990.
- DKOSS84** DeWitt D.J., Katz R.H., Olken, F., Shapiro, L.D., Stonebraker, M.R., Wood, D., *Implementation Techniques for Main Memory Database Systems*, **Proc. ACM-SIGMOD International Conference on Management of Data**, Boston, May 1984.
- Kim80** Kim, W., *A New Way to Compute the Product and Join of Relations*, **Proc. ACM-SIGMOD International Conference on Management of Data**, Santa Monica, June 1980.
- MHWC90** Mohan, C., Haderle, D., Wang, Y., Cheng, J. *Single Table Access Using Multiple Indexes: Optimization, Execution and Concurrency Control Techniques*, **Proc. International Conference on Extending Data Base Technology**, Venice, March 1990. An expanded version of this paper is available as **IBM Research Report RJ7341**, IBM Almaden Research Center, February 1990.
- Moha90** Mohan, C. *ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990. A different version of this paper is available as **IBM Research Report RJ7008**, IBM Almaden Research Center, September 1989.
- PMCLS90** Pirahesh, H., Mohan, C., Cheng, J., Liu, T.S., Selinger, P. *Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches*, **Proc. 2nd International Symposium on Databases in Parallel and Distributed Systems**, Dublin, July 1990, IEEE Computer Society Press. An expanded version of this paper is available as **IBM Research Report**, IBM Almaden Research Center, October 1990.
- SACL79** Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., Price, T. *Access Path Selection in a Relational Database Management System*, **Proc. ACM-SIGMOD International Conference on Management of Data**, Boston, June 1979.
- TeGu84** Teng, J., Gumaer, R. *Managing IBM Database 2 Buffers to Maximize Performance*, **IBM Systems Journal**, Vol. 23, No. 2, 1984.
- Vald87** Valduriez, P. *Join Indices*, **ACM Transactions on Database Systems**, Vol. 12, No. 2, June 1987, pp 219-246.

Appendix A. Key to the Notation

- Avg.Dup.T1': average no. of duplicate join column values in T1
- Avg.Dup.T2: average no. of duplicate join column values in T2
- Avg.Dup.T2': average no. of duplicate join column values in T2'
- COMPHKLK: cost of comparing against high and low keys of an index page
- DataPgs: no. of data pages in a table
- DM: cost for each row examined by data manager
- EXTKEY: cost to extract sort key
- EXTREC: cost of extracting a record after sort
- FF: filter factor of a local predicate
- FFJ12: probability that an outer table row finds a match in the inner table
- FFT1: local predicate filter factor for T1
- LkUnlk: cost for one lock-unlock pair
- NiXLvl: no. of index levels
- NLFpGsT2Ix: no. of leaf pages in T2's index
- NonLfIxPgsT2: no. of nonleaf pages in T2's index
- NpgsT2acc: no. of T2 pages accessed by hybrid join, assuming uniform access
- NpgsWkftT1': no. of pages in the workfiles containing sorted runs T1'
- NpgsWkftT1"T2RID: no. of pages in workfile containing sorted runs T1"T2RID
- NpgsWkft2': no. of pages in workfile containing sorted runs T2'
- NRows: no. of rows in a table
- NRowsT1: no. of rows in table T1
- PFIOPL: cost for initiating a sequential prefetch
- PgMgmt: page pin/unpin, buffer manager, etc. costs per page
- RDS: cost for every row examined by RDS (query processing component)
- RETNRID: cost per RID for returning RIDs to RDS after the first RID
- RetUsr: cost to return a row to the user
- RET1RID: cost of returning the first RID for an index key value to RDS
- SLIDE: cost of slide search in index page
- SqPFIOT: I/O time for a sequential prefetch I/O
- SqPFSz: no. of pages fetched by sequential prefetch in a single I/O
- SSIOTime: skip sequential I/O time assuming uniform distribution over disk
- SSPFIOPL: cost to initialize skip sequential I/O
- SSPFIOSz: no. of pages accessed per skip sequential prefetch I/O
- TTMRG: cost of the tournament tree during merge
- TTSRT: cost for tournament tree during sort
- T1DataPgs: no. of data pages in table T1
- WIOPL: cost to initiate a workfile write I/O
- WIOSz: no. of workfile pages written per I/O
- WIOTime: write I/O time per write I/O
- Wkfdsk: no. of disks supporting workfiles
- WKFFTCH: cost of fetching a row from a workfile
- WkfpGs: no. of workfile pages
- WKINST: cost for inserting a row into workfile

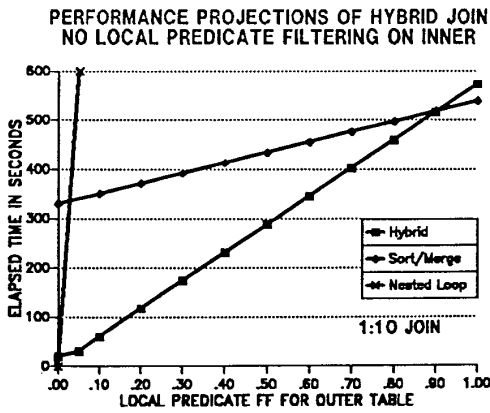


Figure 2: Elapsed Time Projections - 1:10 Join

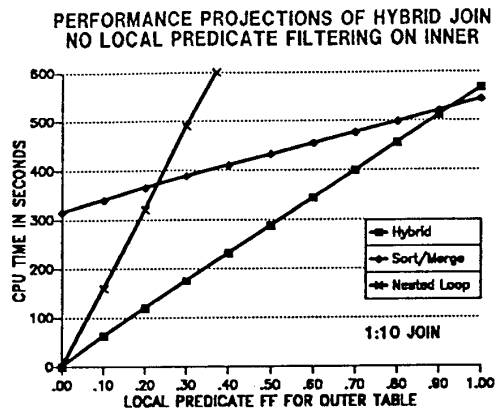


Figure 3: CPU Time Projections - 1:10 Join

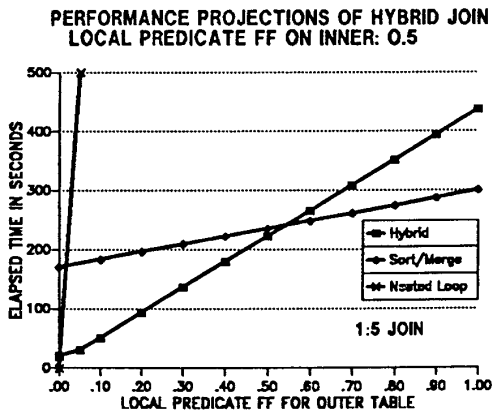


Figure 4: Elapsed Time Projections - 1:5 Join

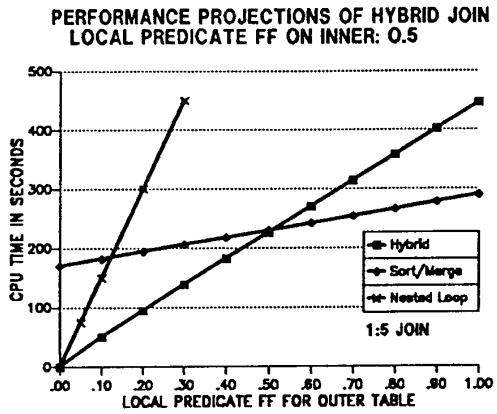


Figure 5: CPU Time Projections - 1:5 Join

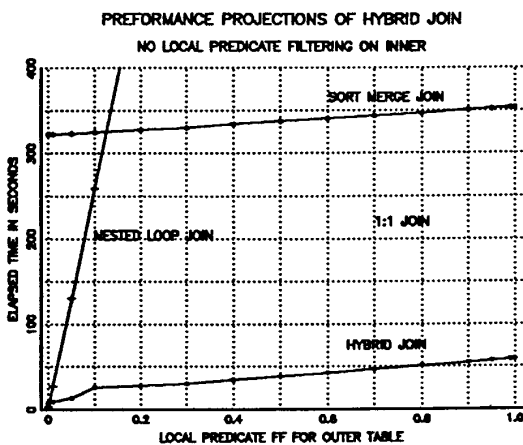


Figure 6: Elapsed Time Projections - 1:1 Join

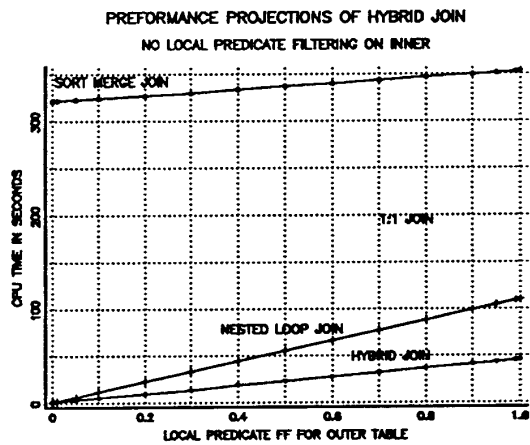


Figure 7: CPU Time Projections - 1:1 Join

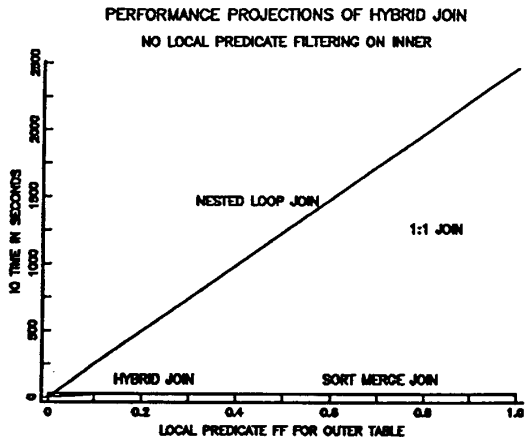


Figure 8: I/O Time Projections - 1:1 Join

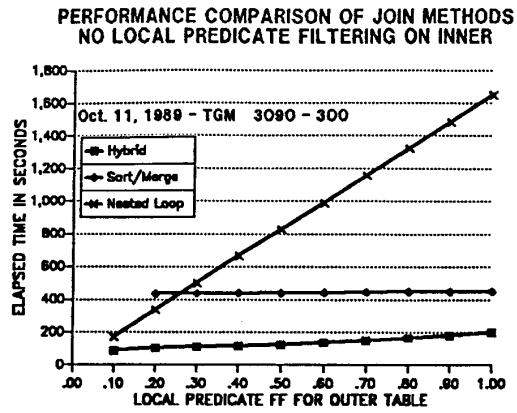


Figure 9: Elapsed Time Measurements - 1:1 Join

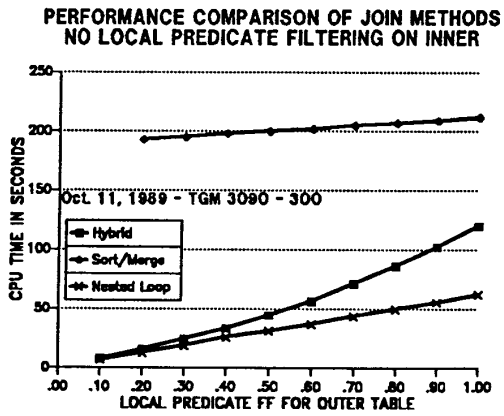


Figure 10: CPU Time Measurements - 1:1 Join

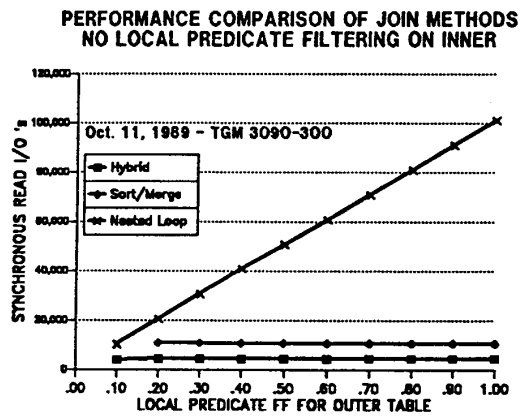


Figure 11: Synchronous Read I/Os Measurements - 1:1 Join