TID Hash Joins

Robert Marek
University of Kaiserslautern, GERMANY
marek@informatik.uni-kl.de

Erhard Rahm
University of Leipzig, GERMANY
rahm@informatik.uni-leipzig.de

Abstract

TID hash joins are a simple and memory-efficient method for processing large join queries. They are based on standard hash join algorithms but only store TID/key pairs in the hash table instead of entire tuples. This typically reduces memory requirements by more than an order of magnitude bringing substantial benefits. In particular, performance for joins on Giga-Byte relations can substantially be improved by reducing the amount of disk I/O to a large extent. Furthermore, efficient processing of mixed multi-user workloads consisting of both join queries and OLTP transactions is supported. We present a detailed simulation study to analyze the performance of TID hash joins. In particular, we identify the conditions under which TID hash joins are most beneficial. Furthermore, we compare TID hash join with adaptive hash join algorithms that have been proposed to deal with mixed workloads.

1 Introduction

Hash Join is a general method for processing equi-joins in relational databases [ME92]. It is especially efficient if the smaller (inner) relation fits completely into main memory. In this case, join processing encompasses two phases. In the building phase, the smaller relation R is scanned and stored in an in-memory hash table by applying a hash function to the join attribute. During the probing phase, each tuple of the outer relation S is read and probed against the hash table. This entails applying the hash function to each join attribute value of S and checking in the corresponding hash class whether there are matching R tuples. This approach has excellent performance since building and probing the hash table can typically be performed with few instructions per tuple. Furthermore, the number of disk I/Os is low since each relation is read only once. Such a hash join provides the most efficient implementation for equi-joins (unless indices on the join attributes exist that can be used for a sort-merge join) [Gr93].

However, if the smaller relation does not fit into memory the performance of hash join typically degrades substantially due to a high amount of additional disk I/O for overflow handling. In this case, the smaller relation is partitioned into p disjoint *partitions* by applying a split function on the join attribute such that each partition fits into memory. The larger relation is partitioned by applying the same split function. Join processing then consists of p smaller joins on the corresponding partitions of the two relations. While there are several alternatives to perform the partitioning (GRACE, hybrid hash join, etc.) they all incur extra I/O. In particular, forming the partitions and storing them on disk may require reading both relations and writing back their partitions. Subsequently, these partitions are read into memory to perform the hash join. Hence, the number of disk accesses may be increased by a factor 3 compared to the case without partitioning. Furthermore, most hash join algorithms require a minimum of $\sqrt{b}+1$ memory pages where b is the number of pages for the inner relation [ZG90]. For very large relations this may limit the applicability of hash joins.

The comparatively poor performance of hash joins in the presence of a limited amount of memory is a main reason why they are not yet widely used in commercial DBMS [Ze90]. Despite the increase of memory sizes, memory-effectiveness is critical for mainly two reasons. First, relation sizes also grow significantly so that in many cases the inner relation cannot be held memory-resident. Second, only a portion of the available memory can generally be used for join processing due to the need to support multi-user processing. Since OLTP transactions typically have higher priority than concurrently executing (large) join queries, a small memory consumption for join processing is very beneficial for supporting mixed workloads.

TID hash joins are a simple and memory-efficient method that aim at avoiding the I/O delays of standard hash joins for overflow handling. Instead of storing entire tuples in the hash table, TID hash joins only store the TID (tuple identifier) together with the join attribute value (key). The TID (sometimes called RID or row identifier) is the physical address of a tuple (page number + offset). The typical TID size is 4 - 8 B allowing much more compact hash tables than with entire tuples. The exact degree of space reduction largely depends on the key size (join attribute size). Assuming “typical” key sizes of 4 - 20 B and tuple sizes of 100 - 1000 B, reduction factors of 4 - 100 can be expected. Hence, for an inner relation of 1 GB (Giga-Byte), a memory size of 10 - 250 MB may be sufficient for a TID hash join to avoid partitioning.

TID hash joins can be based on any standard hash join algorithm. While they try to avoid overflow handling, the reduced space requirements are also beneficial when the compacted hash table cannot be held in memory (see Section 3). On the other hand, TID hash joins require an additional materialization phase at the end to construct the join result. To be effective, the number of I/Os required for this step must be lower than the number of disk accesses saved for overflow handling. This can be expected in many cases since only those tuples need to be materialized that truly contribute to the join result. Furthermore, there are several options to perform these I/O operations

In: **Proc. 3rd International Conference on Information and Knowledge Management (CIKM), November 1994, Gaithersburg, Maryland**

very efficiently (see Section 3). To understand the performance trade-offs associated with TID hash joins, we have constructed a detailed simulation model of a database system. This model enables us to study the behavior of TID hash join algorithms over a wide range of system resource configurations and to conduct a performance comparison with other approaches.

The remainder of this paper is organized as follows. In the next section we briefly discuss some related work. In particular, we describe an adaptive hash join method that has recently been proposed for supporting multi-user workloads. This approach will be used in our performance comparison with TID hash joins. In Section 3, we provide a more detailed description of the implementation of TID hash joins, including several optimizations to limit the materialization overhead. A detailed simulator of a database system that was implemented for studying the performance of the various join algorithms is described in Section 4. Section 5 presents the results of a series of simulation experiments showing that, over a wide range of system and load conditions, TID hash joins outperform standard hash joins and reduce the need for adaptive hash joins. Finally, the main findings of this investigation are summarized in Section 6.

2 Related Work

Performing relational operations (scan, join, sort, etc.) on TID/key pairs is an old idea and in use in several commercial DBMS [Ch91, Gr93]. For instance, sort/merge joins can be implemented very efficiently on B-trees containing key/TID entries if there is such an index on the join attribute for both relations [BE77]. In [Ny93], the implementation and performance of several high-performance sort strategies are described and a key/pointer (=TID) sort is found to be most efficient in many cases. The potential value of TID hash joins was already observed by DeWitt et al. [De84], however without presenting an exact description of such algorithms and without studying their performance.

Recently, several adaptive hash join algorithms have been proposed to support mixed (multi-user) workloads consisting of both join queries and OLTP transactions [ZG90, PCL93]. These algorithms dynamically change the memory allocation for running hash join queries according to the memory requirements of higher-priority transactions. In [PCL93] it was shown that in multi-user mode memory-adaptive hash joins clearly outperform traditional join methods like GRACE and hybrid hash join. The best performance was observed for a new approach called *Partially Preemptible Hash Join* (PPHJ). In the remainder of this section we briefly describe this scheme since it will be used in our performance comparison with TID hash joins.

The PPHJ algorithm is based on a partitioning of the two relations R and S. This gives the required flexibility for changing the memory allocation for join processing by varying the number of memory-resident partitions. The algorithm chooses a fixed number of partitions p with $p = \sqrt{F \times b}$. In this formula, b is the number of pages for relation R and F represents the overhead for the hash table ("fudge factor"). The choice of p constitutes a compromise between a high number of partitions to obtain a high flexibility for changing the memory allocation and large partition sizes to achieve high memory utilization. The PPHJ algorithm is implemented in the following five steps:

(1) *Initialization*
Choose a hash function h that will split R and S each into $p = \sqrt{F \times b}$ partitions, so that each R partition will encompass approximately p pages. Allocate as many R partitions in main memory as the available memory allows. For the remaining R partitions, a single page is allocated as an output buffer. Any leftover memory pages are used as a *spool area* for pages that are being flushed to disk. The spool area is managed by a LRU policy.

(2) *Scan and Partitioning of R*
Scan the inner (smaller) relation R, hashing each tuple using the hash function h. If the tuple belongs to an in-memory partition, insert the tuple into the corresponding hash table; otherwise the tuple is copied to the corresponding output buffer. If an output buffer becomes full, flush it. After R has been scanned, flush all output buffers. They will be needed in the next step to represent S partitions.
In the case that memory has to be taken away from the join, suspend the join if fewer than p pages remain[1]. Otherwise, one or more in-memory partitions have to be flushed. For each affected partition, flush all hash pages and give away all but one of its allocated pages. The remaining page is then used as an output buffer.

(3) *Scan and Partitioning of S*
Scan the outer relation S. Each tuple is hashed with h. If the tuple belongs to an in-memory partition of R, check the corresponding hash table for a match. In case of a match, output the result tuple; otherwise toss the tuple away. If the corresponding R partition's hash table is not allocated in memory, copy the tuple to the S partition's output buffer. Any output buffer that becomes full is flushed. After S has been scanned, flush all output buffers.
If the available memory is reduced during this step, either suspend the join or flush as many in-memory partitions of R as necessary to disk (as in step 2). If additional memory becomes available, bring as many disk-resident R partitions as possible into memory[2]. Future S tuples that hash to these partitions can be joined directly.

In general, the entire inner relation will not fit into main memory. Some R partitions will have to be stored on disk right from the beginning or during steps (2) or (3). For these partitions the corresponding S partitions will be non-empty. To check their S tuples for matches, repeat steps (4) and (5) for each non-empty S partition.

(4) If the hash table of the respective R partition is not already in main memory, read in the R partition and build a hash table for it.

(5) Scan the corresponding S partition, hashing each tuple and probing the hash table. In case of a match, output the result tuple; otherwise drop the tuple.

During steps (4) and (5), disk I/O is avoided for those pages of R and S partitions that still reside in the spool area.

3 Implementation of TID Hash Join

In this section we first sketch how a basic TID hash join can be implemented where the (reduced) hash table for the inner relation fits into memory. Then we discuss several extensions for improving the I/O requirements of this scheme. Finally, we outline the implementation of a memory-adaptive TID hash join algorithm.

3.1 Basic TID Hash Join

An important advantage of TID hash joins is simplicity. If there is already an implementation of a standard hash join algorithm[3] available, only slight modifications are necessary to obtain a TID version of this hash join algorithm. The TID version differs from the standard algorithm in the following ways assuming that no partitioning is necessary:

Modification of the building phase:
In the building phase the inner (smaller) relation R is scanned. Each

1. This is the minimum amount of memory needed for the p output buffers if no R partition can be held in memory.
2. This approach to utilize additional memory was called "expansion" in [PCL93] and was shown to be more effective than alternative strategies.
3. As opposed to **TID** hash joins, traditional join methods storing entire tuples in the in-memory hash-table will be referred to as **standard** join methods.

tuple is hashed and - as opposed to standard hash joins - only TID-key pairs are inserted into the hash table. To obtain reasonable performance this is the minimum amount of information that has to be stored in the hash table. The key is needed to determine matches in the probing phase and the TID is used to access the corresponding tuple in case of a match. Depending on the tuple size in relation to the TID-key pair size a significant space saving can be achieved.

Modification of the probing phase:
In the probing phase the outer relation S is scanned. Each S tuple is hashed with the same hash function used in the building phase and the hash table is checked for a match. In the standard algorithm the matching pair of tuples can be output immediately since both tuples are available in main memory. In the TID version the matching R tuple will most likely reside on disk and has first to be retrieved using the TID. A straight-forward approach would be to directly read the page containing the matching R tuple and to output the matching pair. Performing these I/Os synchronously would introduce enormous delays during the probing phase. As a result, the hash table would have to be kept in main memory for a very long period of time, so that memory-contention may occur in multi-user mode. A better approach is to retrieve matching R tuples during a separate *materialization phase* (see below). This allows a quicker processing of the probing phase and the memory occupied by the hash table can be released much earlier. Furthermore this separation provides us with the possibility to apply some optimizations of the retrieval step (see 3.2).
During the probing phase, matches are recorded in a new *result list* consisting of an (TID_R, TID_S) entry for each result tuple[4]. This list is typically very compact so that it can be kept in main memory. For instance, a single 8 KB page can hold the TID pairs for 1000 result tuples (4 B per TID).

Additional materialization phase:
During this phase the original tuples of the result list have to be retrieved. In the basic version of TID hash join, the TID pairs are read sequentially and for each TID the corresponding tuple is retrieved. While only tuples that occur in the join result are considered, the delays for performing these random I/Os can be substantial.

3.2 Improving I/O performance for TID hash join

There are several possibilities to improve the performance of the materialization phase:

- *Seek optimization (ordered reads)*
  Disk seek delays can substantially be reduced by sorting the TIDs by physical disk location and performing the reads in this order. This also ensures that a page is read only once even if contains multiple result tuples. The seek optimization can be applied to both relations if all result tuples fit into memory. Otherwise, only the tuples of one relation can be read in TID order, while random I/Os remain necessary for reading the matching tuples of the second relation.
- *Keeping qualifying S tuples in the result list*
  The use of TIDs can be avoided for the outer relation S by directly storing qualifying S tuples in the result list. Thus, the result list contains entries of the form (TID_R, matching S-tuple). The result list can be stored in a sequential temporary file for which an output buffer of several pages is maintained in main memory. If the output buffer fills up during the probing phase it is asynchronously written out to disk. In the best case however, e.g., for very selective joins, the entire result list can be kept in memory. In this case no I/O for the S relation is necessary during the materialization phase. For the I/Os on the inner relation R, the seek optimization can be applied as discussed before.
  If the result list was too large to be held in memory, it is read during the materialization phase. This is done by sequential, multi-page read operations that are much more efficient than random I/Os. Furthermore, if the disk device to which the result list was written is equipped with a disk cache, the read I/Os may be served from the disk cache[5]. At any rate, depending on the amount of available memory as many pages as possible should be read from the result list at a time. This is beneficial in order to maximally support the seek optimization for the TID-based disk reads of the matching R-tuples.
- *Use of pipeline parallelism*
  Since probing and materialization operate in a producer-consumer relationship, pipeline parallelism can be exploited between these two phases. In the extreme form, each qualifying S tuple triggers materialization of the matching R tuple by an asynchronous disk read. Ideally, the materialization phase can then largely be performed asynchronously to the probing phase so that materialization does not significantly increase response times. However, the random I/Os necessary for the materialization per tuple can largely reduce the effectiveness of overlapping the two phases, in particular for large join results. Therefore, it may be better to “bundle” multiple entries in the result list before starting their materialization. In this case, several R tuples have to be materialized so that the seek optimization can be applied. Another advantage compared to a sequential processing of the probing and materialization phases lies in the reduced memory requirements for the result list for which no I/Os are necessary any more.
  Of course, overlapping the probing and materialization phases requires that the two relations reside on disjoint sets of disks to avoid disk contention.

The discussion shows that the materialization overhead introduced by the use of TIDs can be kept very low. Extra I/Os on the outer relation S can largely be avoided by storing the qualifying S-tuples in the result list. The I/O delays for materializing qualifying R tuples can also be kept small by performing these I/Os asynchronously during the probing phase. Applying the seek optimization to the R read operations further helps to improve I/O performance for TID hash joins.

3.3 Memory-adaptive TID Hash Join Algorithm

The described modifications for obtaining TID Hash Joins can also be applied if the reduced hash table for the inner relation does not fit into memory and an overflow handling becomes unavoidable. Similarly, it is possible to extend the adaptive hash join method presented in Section 2 to get a memory-adaptive TID hash join (partially preemptible TID hash join).

Since only TID-key pairs are stored in the hash table, the total space requirement of the inner relation is reduced to $F \times b_{red}$, where b_{red} is the number of pages required for storing the TID-key pairs. Assuming a ratio b/b_{red} of 4-100, the number of partitions as well as the average partition size will be reduced by a factor 2-10 for the TID version of the PPHJ algorithm. The lower number of partitions also requires fewer output buffers thus further reducing space requirements. These space savings allow more R partitions to be held in memory, thereby reducing the number of overhead I/Os for writing back R and S partitions during steps (2) and (3) and for reading them in during steps (4) and (5). The result list is constructed during steps (3) and (5) indicating for which R tuples materialization will be necessary. The mechanisms of PPHJ for dealing with memory fluctuations can directly be applied to the TID version of the algorithms. However, due to its lower space requirements it should much less frequently become necessary to reduce the memory allocation for join processing compared to the standard version.

4. Such a result list is similar to a join index [Va87]. However, the join index is a precomputed index structure while the result list is dynamically computed during join processing.

5. Commercial disk controllers keep temporary files in volatile disk caches since these files could be reconstructed after a failure [Gr89].

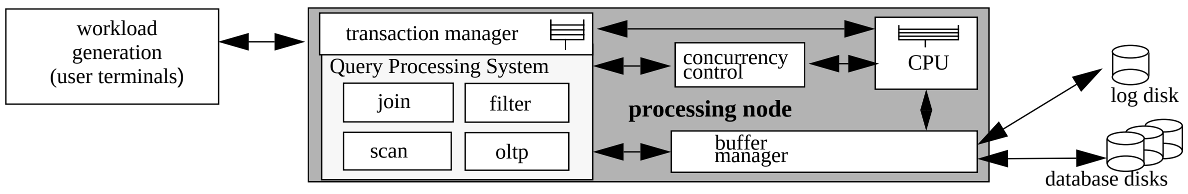

Fig. 1:Gross structure of the simulation system

4 Simulation model

Our simulation system models the hardware and database processing logic of a centralized DBMS architecture[6]. As shown in Fig. 1, the simulation model consists of a *workload generation* component and a *processing subsystem*. We first describe the database and workload models; in 4.2 workload processing is outlined. The simulator is written in DeNet [Li89].

4.1 Database and Workload Model

Our database model supports four object granularities: database, partitions, pages and objects (tuples). The database is modeled as a set of partitions. A partition may be used to represent a relation, a relation fragment or an index structure. It consists of a number of database pages which in turn consist of a specific number of objects (tuples, index entries). The number of objects per page is determined by a blocking factor which can be specified on a per-partition basis. Differentiating between objects and pages is important in order to study the effect of clustering which aims at reducing the number of page accesses (disk I/Os) by storing related objects into the same page. Each relation can have associated clustered or non-clustered B^+-tree indices.

Our simulator allows studying heterogeneous workloads consisting of several query and transaction types. Queries correspond to transactions with a single database operation (e.g., SQL statement). Currently we support the following query types: relation scan, clustered index scan, non-clustered index scan, two-way join queries, multi-way join queries, and update statements (both with and without index support). We also support the debit-credit benchmark workload (TPC-B) and the use of real-life database traces [MR91]. The simulation system is an open queuing model and allows definition of an individual arrival rate for each transaction and query type. The parameter settings of the workload and database model are summarized with the processing system's main parameters in Table 1 (see Section 5.1).

4.2 Workload Processing

The processing component models the execution of a workload consisting of transactions and queries. It comprises a detailed model of the physical resource level and captures the algorithms and techniques used in a relational DBMS. Internally, a processing node is represented by a transaction manager, a query processing system, a buffer manager, a concurrency control component and a CPU server (Fig. 1). The processing node has access to database and log files allocated on external storage devices (disks).

The transaction manager controls the execution of transactions and queries. The maximal number of concurrent transactions is controlled by a multiprogramming level. Newly arriving transactions must wait in an input queue until they can be served when this maximal degree of inter-transaction parallelism is already reached. The query processing system models the processing of OLTP transactions (stored procedures) and the various relational operators. For join processing, several implementations exist including sort-merge and hash join algorithms. For this study, we have implemented the standard PPHJ algorithm described in Section 2 as well the memory-adaptive TID version of this approach (Section 3.3). The TID algorithm uses the basic scheme for materialization described in 3.1 enhanced with a TID sorting for one of the relations (seek optimization). The other optimizations w.r.t. materialization described in Section 3.2 can be approximated (see Section 5).

The number of CPUs and their capacity (in MIPS) are provided as simulation parameters. The average number of instructions per CPU request can be defined separately for every request type. To accurately model the cost of query processing, CPU service is requested for all major steps, in particular for transaction/query initialization, object accesses in main memory (value comparisons, operations on hash tables, etc.), I/O overhead and commit processing. For concurrency control, we employ strict two-phase locking (long read and write locks). Locks may be requested either at the page or object level. A deadlock detection scheme is used to resolve deadlocks.

The database buffer in main memory is managed according to a global LRU (Least Recently Used) replacement strategy. In addition, a memory reservation system under the control of the query processing system allows memory to be reserved in the buffer pool for a particular operator or transaction. In particular, this memory reservation mechanism is used for hash joins to prevent hash table frames from being stolen by other operators. The reservation mechanism is priority-based and allows only higher-priority transactions (e.g., OLTP transactions) to steal frames reserved by a join operator.

Database partitions can be kept memory-resident (to simulate main memory databases) or they can be allocated to a number of disks. Disks and disk controllers have explicitly been modelled as servers to capture potential I/O bottlenecks. Furthermore, disk controllers can have a LRU disk cache. The disk controllers also provide a prefetching mechanism to support sequential access patterns. If prefetching is selected, a disk cache miss causes multiple succeeding pages to be read from disk and allocated into the disk cache. Sequentially reading multiple pages is only slightly slower than reading a single page, but avoids the disk accesses for the prefetched pages when they are referenced later on. The number of pages to be read per prefetch I/O is specified by a simulation parameter.

5 Performance Analysis

The focus of our experiments is to analyze the performance of TID hash joins under various system and load conditions and to compare it with a standard hash join scheme (PPHJ). The most important parameters to consider include the available memory for join processing, the size of the relations, tuples and keys, as well as the join selectivity.

We first provide an overview of the simulation parameter settings used in the experiments. In 5.2 we analyze the performance of the join alternatives in single-user mode. Multi-user experiments for homogeneous and heterogeneous (mixed query/OLTP) workloads are described in 5.3.

6. Although the simulation system supports parallel query processing [RM93], we restrict ourselves to the central case in this study.

Configuration	settings	Database/Queries	settings
no. of CPUs	1	**relation R:**	(100, 200 MB)
CPU speed	20 MIPS	#tuples	250.000
avg. no. of instructions:		tuple size	400, 800 bytes
initiate a query	25000	key size	8 bytes
terminate a query	25000	TID size	4 bytes
I/O	3000	blocking factor	20, 10
read a tuple from memory page	500	index type	clustered B^+-tree
hash a tuple	500	storage allocation	disk
insert a tuple into hash table	100	**relation S:**	(400, 800 MB)
write a tuple into output buffer	100	#tuples	1.000.000
probe hash table	200	tuple size	400, 800 bytes
buffer manager:		key size	8 bytes
page size	8 KB	TID size	4 bytes
buffer size	50 -1600 pages (0.4 -12.8 MB)	blocking factor	20, 10
disk devices:		index type	clustered B^+-tree
number of disk servers	10	storage allocation	disk
controller service time	1 ms (per page)	**join queries:**	
transmission time per page	0.4 ms	access method	via clustered index
avg. disk access time	15 ms	scan selectivity	varied
prefetching delay per page	1 ms	no. of result tuples	1-50 % of the inner relation R
disk cache	20 pages	fudge factor hash table:	1.05
prefetching size	4 pages	arrival rate	single-user, multi-user (varied)

Tab. 1:System configuration, database and query profile

5.1 Simulation Parameter Settings

Table 1 shows the major database, query and configuration parameters with their settings. In this study the workload is composed of join queries and OLTP transactions. The join queries used in our experiments operate on the input relations R and S. The *S* relation contains 1 million tuples, the *R* relation 250.000 tuples. Filter operators performed on *R* and *S* reduce the size of the input relations according to the selection predicate's selectivity (percentage of input tuples matching the predicate). In our experiments, we will reduce the R and S relations to 12.500 (5% selectivity) and 25.000 - 125.000 tuples (2.5 - 12.5%) to be joined, respectively[7]. Both selections employ clustered indices. In order to study the influence of different join selectivities, the join result size is varied between 1 and 50% of the inner relation's size. The result tuples are randomly selected from the two relations.

For all sequential I/Os, in particular relation scans, clustered index scans and scans on temporary files (partitions), prefetching is utilized by the disk controllers. The disk access time for prefetching consists of a base access time per I/O (15 ms) plus an additional delay per page (1 ms). For a prefetching of 4 pages, the average disk access time is 19 ms. Additional I/O delays are incurred at the disk controller and for page transfer.

To capture the behavior of OLTP-style transactions, we provide a workload similar to the debit-credit benchmark. Each OLTP transaction performs four non-clustered index selects on arbitrary input relations (e.g., to branch, teller, account and history records) and updates the corresponding tuples.

5.2 Single-user Experiments

In this section, we compare single-user performance of TID hash join with standard hash join. Memory-adaptiveness is not an issue for single-user mode so that we can assume a fixed amount of memory during join execution. We first present a base experiment to analyze the impact of different memory sizes, tuple sizes and join selectivities. Further experiments consider different relation sizes and potential improvements by an optimized materialization phase.

7. A main reason for assuming such comparatively small relations (as well as small memory sizes) was to keep simulation costs within an acceptable range. However, this is not a significant limitation as the relative performance is mainly dependent on the memory size relative to the relations' sizes rather than the absolute sizes.

Base experiment

For the base experiment, we assumed that the outer relation is ten times as large as the inner relation (125.000 vs. 12.500 tuples after employing the filtering by the index scan). The buffer size is varied between 50 and 1600 page frames (0.4 and 12.8 MB, respectively). While we used constant TID and key sizes of 4 B and 8 B, respectively, two tuple sizes are considered to study the impact of different reduction factors for the TID algorithm. For a tuple size of 400 B (800 B), the TID algorithm reduces the space requirements for the hash table by a factor 33 (66) compared to the standard hash join. The materialization phase of TID hash join performs the R accesses in TID order, but requires random reads for matching S tuples. This turned out to be more effective than ordering the disk accesses w.r.t. S TIDs because there are (10 times) more R result tuples per page than S result tuples.

Figures 2 shows the resulting join response times for the TID and the standard hash join algorithm and the two tuples sizes. For the TID version, the number of matching tuples is varied between 1 and 50% of the size of the inner relation. Since the standard version's performance is not affected by the number of matching tuples, in this case the join selectivity is kept constant (10%). Note that the x-axes specify a relative buffer size which is related to the hash table size for the inner relation in the standard algorithm. Thus a relative buffer size of 100% is sufficient for the standard hash join to keep the inner relation in a memory-resident hash table. For our parameters (page size 8 KB, fudge factor 1.05), this is possible for a memory size of 642 pages (1283 pages) in the case of 400 B (800 B) tuples.

We observe that the standard hash join's performance is very dependent on the available memory, while the TID hash join is more influenced by the join selectivity. For a relative buffer size of at least 100%, the standard version is optimal since there is no need for overflow handling. In this range, TID hash join is always inferior due its need for materializing the join result. However, the performance of standard hash join quickly deteriorates when the amount of available memory is reduced while TID hash join shows almost no performance degradations. This is because, for the standard hash join, reducing the available memory means that fewer R partitions can be held in memory thus leading to increased I/O delays for overflow handling. Furthermore, the standard version is not applicable for less than 51 pages in the case of 400B tuples and less than 75 pages in the case of 800B tuples! The TID version, on the other hand, merely requires 20 pages to store all 12.500 TID/key

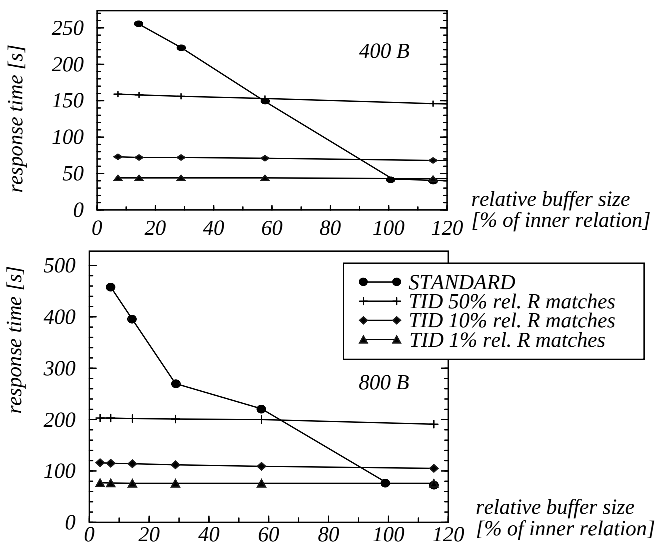

Fig. 2:Influence of buffer size and the join selectivity (12.500 vs. 125.000 tuples)

pairs in main memory. Therefore, no overflow I/O occurs when there are at least 20 pages available. (Note that the TID version still operates with a minimum of 9 pages!). Thus for smaller buffer sizes, TID hash join clearly outperforms standard hash join even for large join selectivities. These observations hold for both tuple sizes, although I/O savings of the TID version are more pronounced for large tuples (800 B).

Since query response times largely depend on the I/O overhead, it is useful to analyze the I/O behavior in more detail. For this purpose, we have summarized the number of overhead I/Os for the different algorithms and configurations in Table 2. Overhead I/Os are all disk accesses in addition to those needed for the initial scans of the inner and outer relations that are needed in any case.[8] For standard hash join, overhead I/Os are needed for temporarily storing partitions of the inner and outer relations on disk due to insufficient memory. For TID hash join, the overhead I/Os are necessary for materialization of the join result. Each entry in Table 2 contains three numbers: the number of overhead I/Os on R, on S and the total number of overhead I/Os. We have specified the absolute buffer sizes since the relative buffer size changes with the tuple size.

The table reveals the extreme I/O overhead of standard hash join for small buffer sizes (for buffer size 50 no join execution was possible). For a buffer size of 100 the number of overhead I/Os of standard hash join is almost twice the number for reading the two relations. This is because almost all partitions had to be written to disk during the partitioning steps and read in later for join processing. For TID hash join, the I/O frequency is primarily determined by the join selectivity which determines the materialization overhead. It turns out that almost all pages of the inner relation have to be read again for materialization except for very selective joins (1%) since there are 10-20 R tuples per page. However, the materialization overhead for TID hash join was more dominated by the I/Os on S. Since we did not apply any optimizations for materializing the S tuples here, the number of overhead I/Os on this relation directly increased with the join selectivity. Furthermore, these I/Os constituted random I/Os. The table shows that increasing the buffer size permits a slight reduction in the number of S I/Os due to buffer hits during the materialization phase.

Despite of the high number of S I/Os with the unoptimized TID hash join, for moderate join selectivities the number of overhead I/Os typically does not exceed the number of I/Os required for the initial scans on R and S. This is always guaranteed for the R relation if we perform the R reads in TID order so that each page is read at most once. The number of S I/Os may only exceed the number of S pages in the case of poor join selectivities so that the number of join result tuples exceeds the number of S pages. The standard hash join, on the other hand, may triple the number of I/Os for small buffer sizes; for a relative buffer size of 50% it still doubles the number of I/Os compared to the best case. Hence, even the straight-forward implementation of TID hash join outperforms standard hash join for moderate join selectivities as long as the relative buffer size is below 50%. This is also confirmed by Fig. 2.

Impact of relation size

The previous experiment already involved two different relation sizes since we varied the tuple size for a fixed number of tuples. We have seen that TID hash join is the more favorable the larger the relations are due to the increased savings for overflow handling. While doubling the tuple/relation size caused a significant I/O increase for standard hash joins, TID hash join experienced only a modest increase in overhead I/Os. The latter was because the larger tuple size only required more I/Os for materializing the smaller R relation, while the number of S I/Os did not increase since the join selectivity remained unchanged.

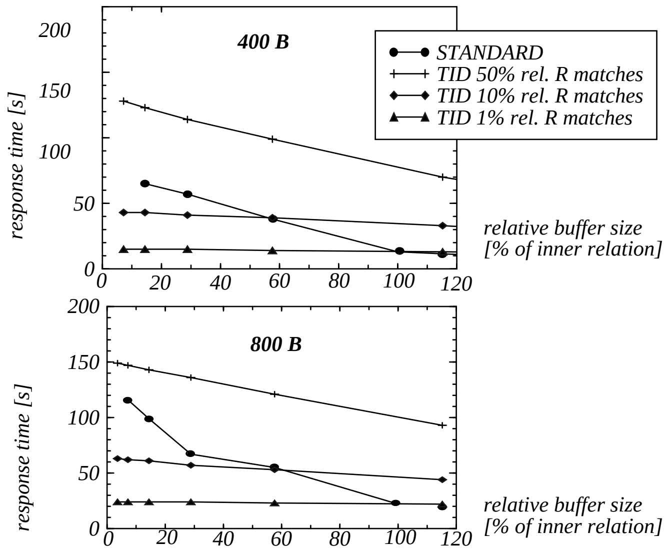

Fig. 3:Influence of the buffer size and the join selectivity (12.500 x 25.000)

We now study the effect when we change the relative size of the two relations to be joined. For this purpose, we left the size of the inner relation unchanged, but decreased the number of S tuples by a factor 5 (12.500 R tuples vs. 25.000 S tuples). The resulting response times for the two tuple sizes are shown in Fig. 3. The main difference to the previous experiment is that standard hash join gains most from the reduced join size. Furthermore, it clearly outperforms TID hash join when 50% of the R tuples qualify, even for small memory sizes. However, this was also due to the fact that the number of join result tuples remained unchanged while the number of S tuples was reduced by a factor of 5. Hence when 50% of the R tuples qualify, 25% of the S tuples had to be materialized as opposed to 5% in the base experiment. Since there are 10 to 20 S tuples per page and the I/Os on S occur at random, each S page had to be read multiple times during materialization explaining the poor performance of TID hash join in this case. However, this problem can largely be avoided by an optimized materialization of S tuples.

Optimized materialization phase

The discussion in Section 3.2 showed that the random I/Os for the materialization of S tuples can be avoided by keeping qualifying S tuples directly in the result list. Furthermore, even I/O delays for the materi-

8. Not counting index accesses, there are 625 (1250) I/Os on R and 6250 (12500) I/Os on S for a tuple size of 400 B (800 B).

join size: 12.500 X 125.000									
tuple	size:	400				800			
algorithm/	result size:	STD	TID 50%	TID 10%	TID 1%	STD	TID 50%	TID 10%	TID 1%
buffer	50	not applicable	625 / 6212 6837	625 / 1244 1869	125 / 125 250	not applicable	1250 / 6228 7478	1250 / 1248 2498	125 / 125 250
size	100	1248/11544 12792	625 / 6155 6780	625 / 1240 1865	125 / 125 250	2738/24331 27069	1250 / 6207 7457	1250, / 247 2497	125 / 125 250
[pages]:	400	619 / 5768 6387	625 / 5857 6482	625 / 1186 1811	125 / 115 240	2072/18446 20518	1250 / 6069 7319	1250 / 1231 2481	125 / 119 244
	1600	0 / 0 0	625 / 4747 5372	625 / 963 1588	125 / 93 218	0 / 0 0	1250 / 5540 6790	1250 / 1133 2383	125 / 103 228

Tab. 2:Average number of overhead disk IOs per join query (on relations R, S and total)

alization of R tuples may be avoided if these disk accesses are overlapped with the probing phase by utilizing pipeline parallelism. Thus, in the best case materialization may not cause any response time increase, especially in single-user mode!

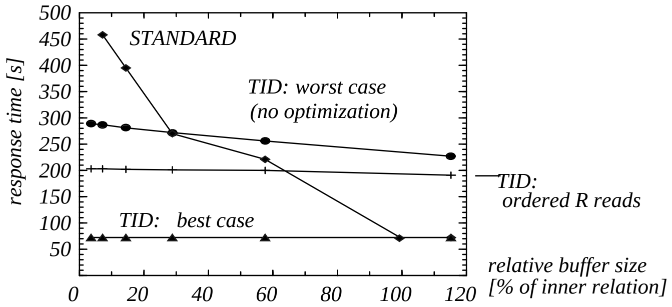

Fig. 4: Impact of different materialization schemes (tuple size 800B; join selectivity 50%)

To illustrate the resulting response time improvements, Fig. 4 shows the performance of TID hash join for three different materialization alternatives. The curves refer to the base experiment (Fig. 2) with a tuple size of 800 B and join selectivity of 50% of the R tuples. In addition to the TID version discussed so far (materialization of R tuples in TID order), the performance of a worst-case and a best-case TID scheme is shown. The worst-case approach refers to the basic TID scheme of section 2.1 with random I/Os on both relations. The best-case assumes keeping the qualifying S tuples in the result list as well as a complete overlapping of probing and materialization phases. Fig. 4 shows that the best-case TID scheme achieves response times equal to those of the standard version without overflow I/Os (100% relative buffer size). This also means that the impact of the join selectivity is largely reduced, greatly expanding the applicability of TID hash join. From the various optimizations, the improved materialization of S tuples is most significant. Interestingly, the worst-case TID scheme still outperforms standard hash join for small memory sizes.

5.3 Multi-user Experiments

To model a base load in multi-user mode, two different workload types are considered in this study: OLTP transactions as well as join queries. Typical OLTP transactions perform a few index selects and update one or more records. Therefore, OLTP transactions tend towards a rather even consumption of the DBMS' physical resources (i.e. disks, main memory and processors). However, since transactions typically require only few main memory pages, disks and processors will more likely become overloaded than main memory. On the other hand, join queries pose relatively high memory requirements when executed using (standard) hash join algorithms. Therefore, running multiple join queries concurrently easily introduces a main memory bottleneck, which in turn may introduce a high number of disk I/Os to cope with memory overflow.

We study the performance of PPHJ and its TID version for heterogeneous and homogeneous workloads. In the former case, we assume a single join query at a time that is concurrently executed with OLTP transactions. The homogeneous workload consists of several concurrently executing join queries. We assume that the memory requirements for OLTP and hash joins are served from the same fixed-size buffer area. Memory requests for OLTP have priority over hash join queries, so that memory may be taken away from running join queries. We assume that each OLTP transaction requests the minimum of one page frame. Furthermore, we assume a join selectivity of 50%, 800B tuples and a large outer relation (125.000 tuples).

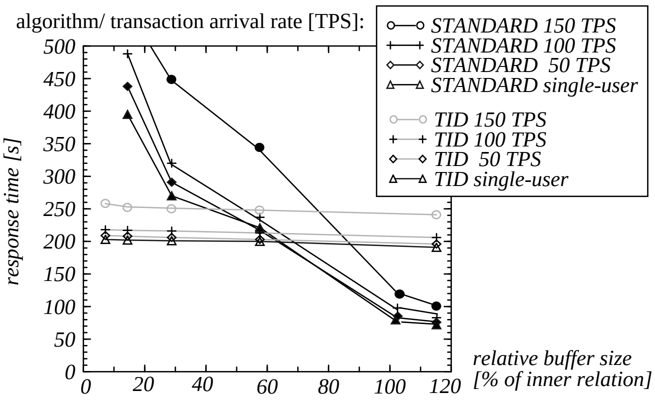

Fig. 5:TID and standard hash join performance in multi-user mode (concurrent OLTP transactions)

We first examine the multi-user results for the heterogeneous workload (Fig. 5). Arrival rates for OLTP are varied between 50 and 150 TPS to obtain different levels of resource contention. Increasing the OLTP arrival rates causes a response time deterioration for both standard and TID hash join due to increased CPU and disk contention. However, for small relative memory sizes the standard PPHJ scheme suffered more under the OLTP load than the TID scheme due to the transactions' memory requirements. While each transaction reserved only a single buffer page, the aggregate memory requirements for OLTP significantly reduced memory availability for join processing in the case of small buffer pools. While there was no need to suspend running queries due to these memory reductions for join processing, the number of overhead I/O increased and thus response times. The TID version, on the other hand, was less sensitive against the transactions' memory requirements. Even in the case of small main memories, concurrent transactions did not increase the joins' I/O overhead significantly. This was because the TID scheme could keep the entire (reduced) hash table of 20 pages in memory.

The small memory requirements of TID joins are even more advantageous in the context of concurrent join queries (homogeneous workload). Figure 6 shows the join response times for this case and different query arrival rates (the mean time between query arrival is varied between 1 and 4 minutes). With growing arrival rates, we ob-

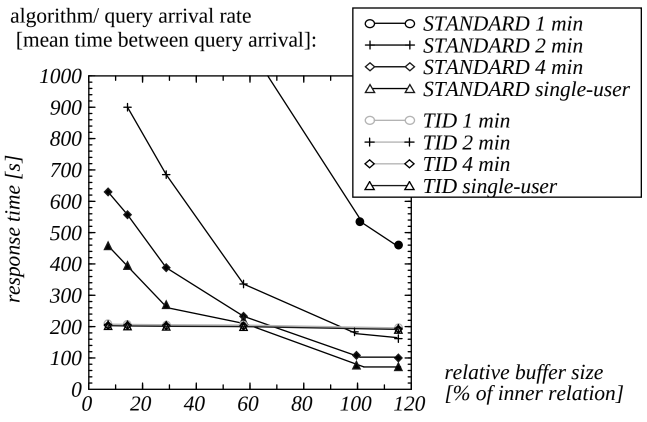

Fig. 6:TID and standard hash join performance in multi-user mode (homogeneous workload consisting of join queries only)

serve a considerable response time increase for standard hash joins. This is because main memory became bottlenecked very soon. Since already running queries occupied a large fraction of the main memory, only few pages remained for newly arriving queries so that they experienced a high number of overhead I/Os. In cases with fewer page frames left than the minimum of 75 pages, new queries had to wait until running queries complete and free memory pages. With higher arrival rates and smaller memory size, most of the queries had to be queued and processed sequentially. Since only few queries could be executed in parallel in these cases, disk and CPU utilization were very low. Therefore, the steep response time increase with growing arrival rates is mainly due to the increased number of overhead I/Os and waits for main memory pages.

Using the TID version, on the other hand, increasing arrival rates affected response times only marginally. Again, this is because of the TID version's low memory requirements. Multiple queries can be processed in parallel without causing resource contention. Concurrent queries neither perform significantly more I/Os, nor force newly arriving queries to wait for memory pages. As a result, response times do not deteriorate, even in the case of high arrival rates.

6 Conclusions

In this paper, we have introduced a simple and memory-efficient method for processing large join queries, namely TID hash joins. TID hash joins are based on standard hash join algorithms but only store TID/key pairs in the hash table instead of entire tuples. Only few modifications are necessary to obtain a TID version from an existing standard algorithm. We presented a detailed simulation study to compare the performance of TID hash join with a memory-adaptive standard hash join scheme (PPHJ). The analysis considered single-user as well as homogeneous and heterogeneous multi-user workloads and identified the conditions under which TID hash joins are most beneficial.

Standard hash joins primarily suffer from the following drawback. If the inner relation does not fit into memory the performance of hash join typically degrades substantially due to a high amount of additional disk I/O for overflow handling. Typically, the number of disk accesses may be increased by up to a factor of 3 compared to the case without overflow. TID hash joins, on the other hand, significantly reduce the memory requirement (typically by more than an order of magnitude). Hence, memory overflow can be avoided in most cases thereby providing significant I/O savings during the building and probing phases. However, they incur extra I/Os for materializing the join results which can be expensive for large join selectivities. To reduce the impact of the join selectivity on the TID hash join's performance, we proposed several materialization alternatives. Most important is avoiding random I/Os for materializing the outer relation's result tuples. This can be achieved by keeping qualifying tuples directly in the result list. Synchronous I/O delays for materializing tuples from the inner relation can also be avoided to a large extent by utilizing pipeline parallelism (overlapping of probing and materialization). Performing tuple accesses in TID order further reduces materialization cost.

Our performance study showed that TID hash join is most beneficial when the inner relation cannot be held memory-resident, in particular if the relative memory size is below 50% of the inner relation's size. It is particularly advantageous in multi-user mode when memory is more restricted due to memory consumption of concurrent transactions and queries. We have seen that TID hash join clearly outperforms memory-adaptive hash join schemes in most multi-user configurations. If multiple join queries are to be executed concurrently, standard hash joins experience very high response times due to a high number of overhead I/Os as well as of frequent query suspension due to insufficient memory. The latter is because standard hash joins require a minimum of $\sqrt{b} + 1$ memory pages where b is the number of pages for the inner relation.

To sum up, TID hash joins are intended to complement rather than substitute standard hash join schemes. In this way, the optimal performance of standard hash join in the presence of sufficiently large memories can be used as well as the superior performance of the TID version for smaller relative memory sizes (e.g., for very large relations) and multi-user workloads. We believe that TID hash join substantially increases the applicability of hash joins and strongly recommend its inclusion in any industrial-strength hash join implementation.

7 References

BE77 Blasgen, M.W., Eswaran, K.P.: Storage and Access in Relational Databases. IBM Syst. Journal 16 (4), 363-377, 1977

Ch91 Chen, J. et al.: An Efficient Hybrid Join Algorithm: A DB2 Prototype. Proc. 6th IEEE Data Engineering Conf. 171-180, 1991

De84 DeWitt, D.J. et al.: Implementation Techniques for Main Memory Database Systems. Proc. ACM SIGMOD, 1-8, 1984

Gr89 Grossmann, C.P.: Evolution of the DASD Storage Control. IBM Systems Journal 28 (2), 196-226, 1989

Gr93 Graefe, G.: Query Evaluation Techniques for Large Databases. ACM Comput. Surveys 25 (2), 73-170, 1993

Li89 Livny, M.: DeNet Users's Guide, Version 1.5. Computer Science Department, University of Wisconsin, Madison, 1989

ME92 Mishra, P., Eich, M.: Join Processing in Relational Databases. ACM Comput. Surveys 24(1), 63-113, 1992

MR91 Marek, R., Rahm, E.: Performance Evaluation of Parallel Transaction Processing in Shared Nothing Database Systems, *Proc. 4th Int. PARLE Conf.* (Parallel Architectures and Languages Europe), Springer-Verlag, Lecture Notes in Computer Science 605, 295-310, 1992

Ny93 Nyberg, C. et al.: AlphaSort: A RISC Machine Sort. DEC Internal Report, June 1993

PCL93 Pang, H., Carey, M.J., Livny, M.: Partially Preemptible Hash Joins. Proc. ACM SIGMOD, 59-68, 1993

RM93 Rahm, E., Marek, R.: Analysis of Dynamic Load Balancing Strategies for Parallel Shared Nothing Database Systems. Proc 19th VLDB Conf., 182-193, 1993

Va87 Valduriez, P.: Join Indices. ACM Trans. on Database Systems 12 (2), 218-246, 1987

Ze90 Zeller, H.: Parallel Query Execution in NonStop SQL. Proc. IEEE CompCon, 484-487, 1990

ZG90 Zeller, H., Gray, J.: An Adaptive Hash Join Algorithm for Multiuser Environments. Proc. 16th VLDB Conf., 186-197, 1990