Hash joins and hash teams in Microsoft SQL Server

Goetz Graefe, Ross Bunker, Shaun Cooper

Abstract

The query execution engine in Microsoft SQL Server employs hash-based algorithms for inner and outer joins, semi-joins, set operations (such as intersection), grouping, and duplicate removal. The implementation combines many techniques proposed individually in the research literature but never combined in a single implementation, neither in a product nor in a research prototype. One of the paper's contributions is a design that cleanly integrates most existing techniques. One technique, however, which we call hash teams and which has previously been described only in vague terms, has not been implemented in prior research or product work. It realizes in hash-based query processing many of the benefits of *interesting orderings* in sort-based query processing. Moreover, we describe how memory is managed in complex and bushy query evaluation plans with multiple sort and hash operations. Finally, we report on the effectiveness of hashing using two very typical database queries, including the performance effects of hash teams.

Introduction

While top-end scalability can be achieved only by database systems that exploit parallel queries and utilities, the vast majority of all database servers have usually many fewer available CPUs than concurrently active requests. For example, one to eight CPUs may serve tens to thousands of concurrent requests. Thus, while intra-query CPU parallelism is important for high-end installations, utility operations, and benchmarks, it is not a panacea for customer applications. Instead, we first have to ensure that sequential query plans can execute as efficiently as possible, and then combine efficient sequential operations into parallel plans when warranted.

Joins and groupings are frequently used operations in relational query processing. Nonetheless, previous releases of the product used only naïve and index nested loops algorithms for joins and an "index nested loops grouping" strategy. While these algorithms are suitable for queries that basically "navigate from record to record" through the database, they can be a severe limitation in data warehouse environments. Moreover, if fast set matching algorithms are available, some query plans may become attractive that did not make sense in the past. Examples include index intersection, union, and difference, as well as joining two indexes of a single table on a common unique column, e.g., the record identifier of the base file, in order to extend the concept of a "covering index" (also called "index-only retrieval") from a single index to multiple indexes. Consider, for example, the simple query *Select A, B From T.* It can be executed by scanning the base file for table *T* or by joining two single-column indexes on columns *A* and *B*. If the two indexes can be scanned faster than the base file, and if the join can be performed very efficiently, the latter plan may be the most efficient one.

For query plans combining multiple indexes as well as for more traditional join plans, we spent considerable effort on implementing the fastest join algorithms possible. In addition to nested iteration, index nested loops, and merge join, we implemented a hash join that is a unique combination of techniques and ideas proposed in the research literature. A description of this hash join implementation is the main subject of this paper.

The implemented techniques include hybrid hashing, dynamic destaging, large units of I/O, partition tuning (also called bucket tuning), recursive overflow resolution with three operator phases (input, intermediate, and output phases similar to input, intermediate, and output phases in sorting), dynamic role reversal between build and probe inputs, histogram-guided skew resolution, "bail-out" using alternative algorithms if inputs contain excessive numbers of duplicates, multi-disk concurrent read-ahead and write-behind, cache line alignment of in-memory data structures, bit vector filtering, and integration into an extensible execution paradigm based on iterator objects. All these techniques are either fully implemented or designed and prototyped. In addition, this is the first implementation of N-ary "teams" of hash joins, which realize in hash-based query processing most of the benefits of *interesting orderings* in sort-based query processing.

If multiple users execute complex query plans with multiple memory-intensive operations such as sorting and hash join, memory becomes a precious commodity, even at today's memory prices, and memory allocation requests on behalf of competing users and of multiple operators within a single query plan must be balanced and coordinated. To the best of our knowledge, there is no known general solution for this problem that considers random arrival and completion of queries, bushy plans shapes, execution phases of complex query plans with multiple "stop and go" operators such as sort and hash join, and, maybe the most difficult yet most frequently ignored issue, the notorious inaccuracy in estimating the sizes of intermediate results. Given that we had to implement some

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

solution to this problem, even if we cannot claim or even prove optimality, we designed what we believe is a pragmatic and robust solution, which is also described in this paper.

Related work

Since we are working to ship a robust and maintainable product, we could not afford to undertake or use very much unproven research. Our contribution has been to integrate and to innovate in integration rather than to innovate in specific techniques. Many relevant papers have been summarized and cited in an earlier survey [Graefe 1993]. While hybrid hash join and parallel hash join are well-known ideas today [DeWitt 1984, DeWitt 1985], our appreciation for the insights by Bratbergsengen, Sacco, and Kitsuregawa and their collaborators [Bratbergsengen 1984, Sacco 1986, Kitsuregawa 1989] on hash-based query evaluation methods has grown since the writing of that survey, e.g., partition tuning [Kitsuregawa 1989]. Some of the techniques combined in our implementation, in particular histogram-guided skew management and N-ary hash join, have been vaguely described in an earlier paper, which also provides a rationale for preferring hash join over merge join in very many cases yet for implementing both join algorithms in a product [Graefe 1994].

Among competing products, Tandem's hash join is one of the few commercial implementations described in some detail in the open literature [Zeller 1990]. Oracle uses hybrid hashing for joins, including dynamic role reversal, but not for grouping and duplicate removal. Informix also uses hybrid hashing, but we believe that this implementation is not as sophisticated as the hashing algorithm described here. Red Brick employs hybrid hashing with multi-level recursion, role reversal, and bit vector filtering [Red Brick 1996]. NCR's Teradata product uses hashing extensively, including partitioning and merge join on hash values as well as hash grouping. IBM's DB2/400 employs hashing for grouping and for joins, including bit vector filtering, and relies on the AS400's single-level store instead of hybrid hashing, partition tuning, and recursive partitioning. IBM's DB2/CS team prototyped a hash join implementation, but decided not to include it in the product. Sybase has implemented a hash join in the Sybase IQ product and is rumored to be implementing a hash join in its main relational database product.

The benefits of hash join and hash grouping

Implementing hash join and hash grouping operations can be quite complex. Given that most database systems already have nested loops join and merge join algorithms as well as sort- or index-based grouping, what value do hash-based algorithms add? The truth is that there is no one overall winner, and each of these algorithms is superior to its alternatives in certain circumstances.

Since hash-based algorithms process large, unsorted, non-indexed inputs efficiently, they are particularly useful for intermediate results in complex queries, for two reasons. First, intermediate results are not indexed (unless explicitly saved to disk and then indexed) and often are not produced suitably sorted for the next operation in the query plan. Second, since query optimizers only estimate intermediate result sizes, and since estimates can be wrong by more than an order of magnitude in complex queries, algorithms to process intermediate results must not only be efficient but also degrade gracefully if an intermediate result turns out to be much larger than anticipated.

Once an efficient join algorithm is available that requires neither sorted nor indexed inputs, some of the query plans traditionally associated with bitmap indexes can be exploited, such as index intersection, difference and union. Bitmap indexes and other types of non-unique non-clustered indexes differ in their representations of the set of row identifiers associated with each search key. There are multiple well-known representations of sets in computer programs; bitmap indexes represent each such set as a bitmap, whereas conventional database indexes explicitly enumerate all members of each such set. All operations possible with bitmap indexes are also possible with conventional indexes; for intersection, difference and union, hash join is typically the algorithm of choice.

Efficient join algorithms also open two further avenues of database performance improvement. First, semi-join reductions may apply not only to distributed but also to single-site database systems. Second, in many databases, de-normalization is used achieve better performance. In other words, in order to save join operations, databases are designed that violate the well-known normal forms, in spite of the dangers of redundancy such as inconsistent updates. On the physical level, i.e., on the level of abstraction describing disk space and indexes and query execution plans, de-normalization is often called "master-detail clustering." It is usually explained with orders and their line items, and is a viable and effective technique to improve database performance. However, on the logical level, i.e., on the level of abstraction describing tables and constraints and queries, de-normalization is a bad idea: the dangers of redundancy and the virtues of normal forms are well known. With efficient join algorithms, including hash join, the motivation for de-normalization diminishes or even vanishes. Taken to the opposite extreme, vertical partitioning (representing groups of columns from a single table in separate files or indexes) may become a viable option for physical database design. We'll come back to this point in the performance evaluation.

An overview of hash join techniques and terminology

For completeness, we describe the basic ideas of hash-based query processing. More detailed descriptions, algo-

rithm variants, cost functions, etc. can be found elsewhere, e.g., [Graefe 1993].

As a join operator, a hash join has two inputs, which are called "build input" and "probe input," for reasons that will be apparent shortly. The optimizer assigns these roles such that the smaller one among the two inputs is the build input.

Our hash join implementation implements many types of set matching operations: inner join; left, right, and full outer join; left and right semi-join; intersection; union; and difference. Moreover, with some modest modifications, it can do duplicate removal and grouping (like "sum (salary) group by department"). These modest modifications boil down to using the one and only input for both the build and probe roles. We explain inner join only in this section, and leave it to the reader to adapt the description to the other operations.

Hash join (like merge join) can only be used if there is at least one equality clause in the join predicate. This is usually not an issue because joins are typically used to reassemble relationships, expressed with an equality predicate between a primary key and a foreign key. Let's call the set of columns in the equality predicate the "hash key," because these are the columns that contribute to the hash function. Additional predicates are possible, and are evaluated as "residual predicate" separately from the comparison of hash values. Note that the hash key can be an expression, as long as it can be computed exclusively from column in a single row. In grouping operations, the columns of the "group by" list play the role of the hash key. In set operations such as intersection, as well as in duplicate removal, the hash key consists of all columns.

There are several cases that are interesting to distinguish, based on input sizes and memory size (for grouping and duplicate removal operations, the output size also matters). The simplest case is defined by a build input that is smaller than the memory allocated to the hash join operation. In that case, the hash join first consumes (scans or computes) the entire build input and builds a hash table in memory (hence the name "build input"). Each record is inserted into a hash bucket depending on the hash value computed for the its hash key. Because in this case the entire build input is smaller than the available memory, all records can be inserted into the hash table without any problem. This "build phase" is followed by the "probe phase." The entire probe input is consumed (scanned or computed) one record at a time, and for each probe record, the hash key's hash value is computed, the corresponding hash bucket scanned, and matches produced.

If the build input does not fit in memory, a hash join proceeds in several steps. Each step has two phases, build phase and probe phase. In the initial step, the entire build and probe inputs are consumed and partitioned (using a hash function on the hash keys) into multiple files. The number of such files is called the "partitioning fan-out." In other words, the two inputs are partitioned into "fan-out"-many pairs of files. Using the hash function on the hash keys guarantees that any two joining records must be in the same pair of files. Therefore, the problem of joining two large inputs has been reduced to multiple instances of the same problem, but of smaller size – a prototypical "divide-and-conquer" algorithm. In other words, we simply apply the same algorithm again to each pair of partition files – with a modified hash function, of course.

In the worst case, multiple partitioning steps and multiple partitioning levels are required. Note that this is required only for very large inputs, i.e., inputs for which a standard external merge sort would require multiple merge levels. Also, if only some of the partitions are very large, additional partitioning steps are used only for those. In order to make all partitioning steps as fast as possible, large, asynchronous I/O operations should be used, such that a single thread can keep multiple disk drives busy with useful work.

The first case above is called "in-memory hash join;" the second case, "Grace hash join" (after a database machine research project [Fushimi 1986]); the third case, "recursive hash join." If the build input is somewhat larger but not a lot larger than the available memory, it is possible to combine elements of in-memory hash join and Grace hash join in a single step – this is called "hybrid hash join." Basically, some hash buckets are build and probed in memory, whereas records belonging to all other hash buckets are spilled to partition files (also called overflow files).

Often, an optimizer's estimate of the count of incoming records is off by a factor or 2 or even 10, in either direction. Thus, it is not always possible during optimization to determine reliably which of these cases will occur at run time. Our implementation (like most others) starts as in-memory hash join, but gradually transitions to hybrid, Grace, and recursive hash join if the build input is very large. This has been aptly called "dynamic destaging" in the Grace papers, but that name is not universally known or used.

If a hash bucket has been spilled in the build phase of any one step, our implementation retains in memory a bit vector filter. For each build record spilled to an overflow file, the hash value of the hash key is used to determine a bit to set "on." In the probe phase of that step, the bit vector is checked to determine whether a probe record at hand can possibly have a match in the build input. If not, the probe record can either be discarded (e.g., in an inner join) or produce an output record (e.g., in a full outer join) – in either case, the probe record doesn't go to the probe overflow file and therefore doesn't incur I/O costs. This technique is called "bit vector filtering" and can be quite effective – we believe several hash join implementations employ this performance enhancement.

If the optimizer anticipates wrongly which of the two inputs is smaller and therefore should have been the build input, or if bit vector filtering has reduced the probe overflow file for some partition so much that it is smaller than its corresponding build overflow file, the two roles (build & probe) can be reversed in the next step. In other words, before

processing a spilled partition, our hash join makes sure that it uses the smaller overflow file as build input. This technique is called "role reversal" – many hash join implementations use this technique, too.

Hash join implementation

Data structures

Let us begin the description of our implementation with an overview of its core data structures. The hash table is a large array, say 1,000 or 50,000 slots, depending on the size of the available memory. Each slot anchors a linked list, which we call a hash bucket. Each element in a linked list contains a full four-byte hash value and a pointer to a record pinned in the buffer. Note that we use buffer space to retain records in memory; in other words, the hash table proper contains pointers to records, but not actual records. Multiple elements in a bucket are allocated together, in units of 64 bytes, to optimize the faulting behavior in CPU caches by reducing the number of *next* pointers in the linked lists. When counting or limiting the memory space to be used by a hash join, we count both the elements in the lists and the buffers containing the records.

Each bucket is assigned to a partition[1]. There is an overflow file for each partition and for each input. Thus, a binary operation such as a join with a partitioning fan-out of 10 will have 20 overflow files, although some of them, in particular probe overflow files of resident partitions, will be empty and will not have any disk or buffer space allocated for them. While a partition is *resident*, all its records from the build input are pinned in the buffer, accessible through their respective hash buckets. When a partition is *spilled*, its records are not accessible through the hash table, and hash table slots for the partition's buckets are used not as pointers (anchors of linked lists) but as bit vector filters. Knowledge and control over which partitions are currently spilled is delegated to a separate object, in order to facilitate "teams" of multiple hash operations for N-ary operations, as described later.

Each partition includes one overflow file for each of the operation's inputs – for example, a unary operation such as duplicate removal has one input, a join has two inputs, and an N-ary operation has N inputs such that each partition is represented as an N-tuple. There are three sets of overflow partitions. The first set contains partitions that are currently being built by the current partitioning step. In hybrid hash join, some of these partitions may be resident, while others are spilled. After each partitioning step, this set is emptied and its members discarded or distributed to the other two sets. The second set contains partitions that are ready for final processing – those partitions will be processed with a hybrid hash join step, without recursive overflow resolution. The third set of partitions contains those partitions that are too large for hybrid hashing, i.e., those partitions that require a partitioning step with full fan-out and recursion. In many practical situations, this set remains empty throughout execution of a hash operation.

In addition to these data structures that capture and retain the state of a hash operation, there are also data structures with control information, e.g., whether the present operation is an inner or an outer join, which is probably explained best while describing the algorithm itself.

Basic algorithm

The basic components of the hash join algorithm consists of three nested loops. The outer-most loop iterates over partitioning steps, the second loop iterates over the inputs, and the inner-most loop iterates over records. We try to explain these loops in outer-to-inner order.

A partitioning step is a general term that can be either the initial step that consumes the hash operation's inputs or an overflow resolution step that consumes overflow partitions created in previous steps, and it can take the form of a traditional in-memory hashing operation with no overflow records being written to disk, a hybrid hash join with some partitions resident in memory and some partitions spilled to overflow files, or a full partitioning step with no in-memory hash table. A partitioning step determines its partitioning fan-out based on the available memory, creates an empty hash table, processes all inputs, drains and destroys the hash table, and disposes of all partitions. Partitions are either discarded directly without incurring any I/O (those that were resident at the end of the partitioning step, and are therefore completed) or moved to another set of overflow partitions, either the set ready for final processing or the set requiring full partitioning steps. This latter decision is considered in more detail later.

The core of each partitioning step is a loop over the operation's inputs. This loop permits using a single code base for both unary operations such as grouping and duplicate removal as well as binary operations such as inner and outer joins. Moreover, the loop avoids multiple copies of a fair amount of code and, more importantly, of complexity and of code maintenance. An incidental but not unimportant benefit is that it is trivial to support binary operations such as joins with grouping or duplicate removal on the build input, which actually is useful quite frequently because duplicate records can safely and beneficially be removed for semi-joins and, interestingly, many group-by clauses are foreign keys to be used in a join following the grouping operation. Finally, as a matter of curiosity, the basic algorithm does not require that there be at most two inputs, although we currently do not exploit this capability. Candidate operations that could exploit this capability such as *union distinct* of more than two inputs are mapped to alternative plans using concatenation of intermediate results, and N-ary operations are implemented using "teams" of multiple unary and binary operations. The role played by a specific input, i.e., build or probe input, is rep-

[1] While some other authors use the term *bucket* where we use *partition*, we have chosen to use *bucket* for in-memory division and *partition* for dividing inputs into multiple overflow files on disk.

resented by a small set of control values, in particular for searching in the hash table, aggregation or insertion into the hash table, output to the next operator in the query plan, and for setting or testing the bit vector filter.

For each input, the algorithm opens the record source, which is an operator in the initial partitioning step or an overflow file in a subsequent partitioning step, creates and opens a suitable number of output overflow files, loops over the input records, and closes record source and overflow files. Processing an input record begins with determining its hash value, hash bucket, and partition. If the partition is resident and search is required, the hash bucket is searched for matches with the current input record. If insertion into the hash table is required, possibly based on the outcome of the search, a record slot is allocated in the partition file and a new linked list element is inserted into the hash bucket. If output is required, also possibly depending on the search, an output record is produced and passed to the next operator in the query plan. If the input record's partition is not resident but spilled, the bit vector filter may be set or tested, the record may be appended to the partition file, or an output record may be delivered (in outer joins, based on the bit vector filter).

Hybrid hashing and dynamic destaging

At the beginning of a step, all buffers assigned to the hash operation are considered available, and we ensure that the number of buffers is always equal to or larger than the number of partitions. While a resident partition may pin records on and thus occupy an arbitrary number of buffer pages, a spilled partition uses only one output buffer at a time; write-behind buffers are allocated for the entire partitioning step as a whole and are not counted here. When a resident partition requires a new page, one of the available buffers is assigned to the partition. If no buffer is available, the largest resident partition is spilled. Note that the largest resident partition occupies at least two pages. When the largest resident partition is spilled, either some page is freed, or the partition requiring an additional page is spilled, in which case its memory requirement drops to a single page. The second justification for spilling the largest resident partition is that, in the absence of any knowledge about skew in the probe input but presuming that the probe input is larger than the build input, spilling the largest build partition results in the best ratio of saved I/O to required memory.

Setting the partitioning fan-out as a function of the buffer size is not simple, because input size estimation may be inaccurate even by orders of magnitude for very complex queries. First, some buffers are taken off the top to be used for asynchronous read-ahead and write-behind. In general, it doesn't pay to have more buffers for asynchronous I/O than disks, because each disk can perform only one I/O action at a time. The remaining buffers can be used either to increase the fan-out or to increase the size of the in-memory hash table. If the build input is particularly large and recursive partitioning will be required, the fan-out should be as large as possible. On the other hand, if the build input is only a small multiple of the memory size, a large fan-out requires more output buffers than necessary and therefore over-restricts the size of the hash table for the resident partitions. Our pragmatic compromise is to set that fan-out for the initial partitioning step such that about 50-80% of available buffer space might be used, depending on our input size estimate as well as our confidence in that estimate. Partition tuning at the end of the build phase can alleviate the effects of a fan-out chosen too large and a hash table chosen too small.

More important than the exact fan-out is the size of each I/O unit. In order to achieve a sustained bandwidth of about 50% of the disks' raw transfer speed with random I/O as needed in partitioning, the data volume for each disk access has to fairly large; pragmatically, we chose 64 KB as our unit of I/O using scatter-gather I/O of in-memory pages of 8 KB, our system page size. In general, the goal is to maximize the product of sustained bandwidth and the logarithm of the fan-out, because we believe that a merge step in external merge sort and partitioning in hash-based algorithms are very closely related, and because this goal maximizes the number of comparisons per unit time in a merge step.

Partition tuning

For binary operations such as join (as well as N-ary operations), partition tuning at the end of the build phase may save substantial I/O for probe overflow files. If some of the spilled build overflow files are smaller than the memory allocated for the hash operation, multiple spilled partitions may be processed concurrently, and it is sufficient to create a single probe overflow file for an entire group of build overflow files. Thus, output buffer space may be saved during the probe phase, and the hash table size may be increased. In our implementation, we employ a simple first-fit heuristic bin packing algorithm on the spilled build overflow files, and restore the build overflow file of as many partitions as possible into memory, restoring the smallest build overflow file first.

While this technique is quite effective, in particular if the build input size is a small multiple of the available memory size, recursive overflow resolution, as required for very large inputs, can use an even more effective method, based on histograms on hash value distributions.

Recursive overflow resolution, three phases, and iterator methods

As mentioned above, there is a loop over partitioning steps, and each partitioning step reads from the input operators (first partitioning step) or from an overflow partition (all subsequent steps) and writes zero or more overflow partitions to disk. The important point is that the recursion in multi-level partitioning, which is a direct dual to multi-level merging in external sorting, has been rewritten into a loop using well-known recursion-to-loop transformation

techniques using a stack maintained explicitly (see, e.g., [Sedgewick 1984]). Unfortunately, this is not quite sufficient to integrate recursive partitioning into a single-thread query execution engine, because the hash operation still has to fit into the scanning or iterator paradigm, i.e., respond to *open*, *next*, and *close* methods[2]. To reduce the complexity in the algorithm, we retained the three explicit loops and adapted the standard implementation of co-routines. We map the *open*, *next*, and *close* methods to a single method, retain all state not on the general call stack but in a special state object that retains its contents between invocations of the iterator methods, including the program counter, and jump to the program counter each time the co-routine is resumed. Instead of an explicit program counter, we combine a variable of an enumerated data type with a *switch* statement using a *go to* statement in each *case*, all encapsulated in suitable macros. This technique can be used both on the input and output side of iterators, thus creating data- or demand-driven iterators, even within a single thread, which can be very useful to "push" the result of a common sub-plan to multiple consumer iterators.

Overflow files themselves might be very large. In that case, all overflow partitions are divided into two sets, depending on whether processing an overflow partition will require that all sub-partitions be spilled because they, too, are larger than memory, or will result in hybrid hashing with some sub-partitions remaining resident in memory. This point is sometimes referred to as "memory squared," despite the lack of attention to the difference between fan-out and memory size due to large units of I/O, buffers for asynchronous I/O, etc. If there are partitions of such large size, it is important to employ the largest possible fan-out while partitioning those. During the initial partitioning step, the hash operation competes for memory with its input iterators and plans; once the first output item has been produced, the hash operation competes with its consumer iterator and plan. However, if the inputs are truly very large, it is useful to ensure that some partitioning steps can use the entire memory allocated to the query plan, without competing with any other iterators in the same plan. In our implementation, if none of the previous partitioning steps has produced output, i.e., the present hash operation is still in its *open* method, any overflow partition larger than the limit for hybrid hashing is assigned to a special set of overflow partitions. When choosing the input partition for the next partitioning step, partitions in this set are preferred over all other waiting partitions. Moreover, those partitioning steps set the fan-out to use 100% of the available buffer space as output buffers (excluding buffers used for read-ahead and write-behind), and spill all sub-partitions right from the start, without waiting for dynamic destaging. It may be interesting to note that partitioning steps that do not compete with either input nor output plans for memory and may therefore use all memory available to the query to maximize their partitioning fan-out are a direct dual to the intermediate merge steps in external sorting for very large inputs.

[2] We extended this triple with some additional ones, e.g., *rewind* and *rebind* (with new correlation values or parameters). The hash operation simply restarts for any of these.

Role reversal

Given that selectivity estimation can be rather inaccurate, and that hash joins derive much of their advantage over merge joins from their asymmetry, i.e., from using the smaller of the two inputs as build input, a natural idea is to defer the assignment of build and probe roles to the two inputs from compile-time to run-time. Moreover, even if the two inputs are equal in size, bit vector filtering during the initial partitioning step may result, for some or all spilled partitions, in probe overflow files smaller than the corresponding build overflow files. Finally, hash value skew may also result in this situation for some but not all spilled partitions. Therefore, when starting to process a spilled partition, our implementation always considers role reversal. Note that in recursive partitioning, in particular if bit vector filtering is effective, role reversal back and forth may be useful in multiple successive recursion levels.

Bail-out

If, after several recursion levels, none of the techniques above results in build overflow files small enough to permit in-memory or at least hybrid hashing, both input files probably contain so many duplicate keys that no partitioning method can succeed. In this case, our implementation resorts to sort- or loops-based join and grouping algorithms. The required alternative execution plans are created at compile-time using deterministic algorithms, without optimizating plan search. Note that these algorithms are applied to individual overflow partitions, not the entire input. Note also that this case is very rare, and therefore it is not performance-critical – the primary concern is that the problem be resolved correctly and in all cases, with the reasonable development and testing effort.

There are very few alternative bail-out strategies. Other than sort- and loops-based strategies, one can resort to additional memory allocations, data compression, or dropping columns from overflow files. Additional memory grants can disrupt the entire server, as well as lead to deadlock (waiting for a memory grant while holding a lock on data). Data compression, in particular data compression on the fly, is very complex and would have introduced substantially larger implementation and testing effort. Dropping columns from partition files (replacing them with pointers into permanent files, and re-fetching the dropped column values later) can be very expensive due to the cost of fetching. Most importantly, however, all three alternative strategies only alleviate but do not completely resolve the problem. If the set of duplicates is truly very large, these three alternative strategies might well fail. Our goal, however, was to find a robust and complete solution for this rare case.

Histogram-guided partitioning

One of the reasons why recursive partitioning might require multiple levels is skew in the hash value distribution, both distribution skew and duplicate skew. In order to avoid the worst cases, we employ histogram-guided recursive partitioning, i.e., while writing a set of partitions, we gather distribution statistics for the next recursion level. In fact, if the partitioning fan-out is *F*, we gather statistics for as many as *4F* future sub-partitions for each output partition currently being created. Before starting to read such a partition, a first-fit heuristic bin packing algorithm is used to create either *F* real partitions of 4 sub-partitions on average, or fewer partitions with build overflow files barely smaller than memory, plus one partition fitting into the memory not required for output buffers. The histograms can also be used to plan for role reversal as well as to decide earlier, before rather than after wasting an entire partitioning step, when to switch to a sort- or loops-based algorithm because partitioning cannot divide an excessively large set of duplicates into sets smaller than memory.

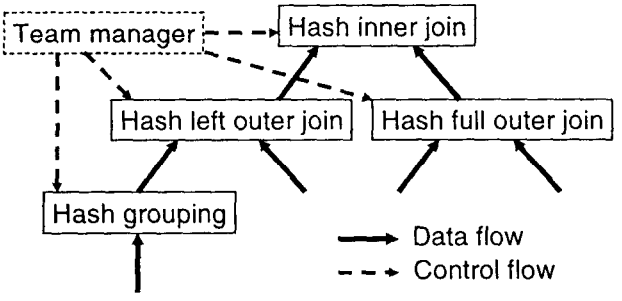

Figure 1 – Data and control flow between operators and team manager

Teams for N-ary hash joins

The most innovative techniques in our hash join implementation is the notion of "teams," i.e., multiple hash operations cooperating to process their inputs, often more than two inputs. These operations can be any of the implemented set matching operations, i.e., joins, grouping operations, etc., in any combination, even bushy plan segments. Teams are used if multiple neighboring operations in a query plan hash on the same column(s), and ensure that intermediate results between members of the same hash team do not have to be partitioned by the consumer operator. For example, if three tables are to be joined on a single common column, a traditional pair of binary hash joins will partition four inputs, namely the three original tables as well as the intermediate result, whereas a team will incur overflow I/O only on the three original inputs.

Operators' tasks	*Team manager's tasks*
Consume input records	Map hash values to buckets
Produce output records	Map buckets to partitions
Manage hash table	Grant memory requests
Manage overflow files	Request to spill & to restore from entire team
Request memory grants	
Spill partitions on request	
Restore partitions on request	

Figure 2 – Tasks of operators and team manager

In effect, teams are a dual to *interesting orderings* in sort-based query processing, which permit omitting an intermediate sort between joins or grouping operations on the same keys. Interesting orderings have been well recognized as important to exploit in query optimization [Selinger 1979]. Using the same example of joining three tables, if two merge joins use the same join key, only three sort operations for the three original tables are required, whereas four sort operations are required if the two joins use different keys and therefore require different sort orders.

The essential point is that intermediate sort operations can be saved if neighboring operations in a query execution plan require and produce ordering on the same sequence of columns. This idea is well known ever since the classic query optimization paper [Selinger 1979], and has been considered an important technique in all database query optimizers ever since. Given that teams apply in the same cases as interesting orderings and basically have the same effect, and that most query optimizers attempt to exploit interesting orderings, among the query processors employing hash-based joins and grouping (in commercial as well as research database systems), ours is the only query processor to implement and exploit teams.

The fundamental idea of N-ary hashing or "hash teams" has been vaguely described in the past [Graefe 1993, Graefe 1994]. The key to implementing hash teams is to separate control over spilling (overflow) and partitioning policy decisions from the individual hash operation and to assign it to a team manager, which performs this control function for all member operations within the team. As a result, all team members (which, by definition of a team, hash on the same set of columns) spill, restore, and process the same partitions at the same time, even when recursive partitioning is required. The division of tasks between the team manager and the team members, i.e., operators such as hash inner join, is given Figures 1 and 2. The important effect is that, among two members of a team that form a producer-consumer relationship, the consumer can be sure that the producer won't produce any data items for a partition once that partition has been spilled. Thus, a team avoids partitioning intermediate results between team members, and saves all (or actually almost all) I/O to and from overflow files for intermediate results between members of the same team. Moreover, the consumer may release all memory, including all output

buffers, for a spilled partition. Thus, for a spilled partition, only one output buffer at a time is required within the entire team, exactly as for the two inputs in a binary hash join.

It is possible that a partition is spilled after some records have been forwarded from a producer to a consumer within the same team. Thus, the consumer must allocate an overflow files for all of its inputs, even those that are members of the same team. When a partition spills during the consumer's build phase, the records already in the consumer's hash table are spilled and the build overflow file for that partition may be closed without retaining an output buffer. When the spilled partition is processed in a later partitioning step, both the overflow file and the input iterator's further output for that partition must be consumed. Thus, the loop to consume all records in an input partition has been augmented to read first any overflow file and then, if the input is a member of the same team, to consume any remaining input for the current partition.

Unfortunately, teams can inhibit other performance enhancements in some special cases. For example, consider the case that the producer in a team is a full outer join (but not inner join or semi-join). Since bit vector filtering might produce output for spilled partitions, bit vector filtering must be switched off in the producer. Of course, bit vector filtering can be used in the root operator of the team, even if that operator is an outer join.

What's missing

Among the techniques described, three are designed and prototyped but not implemented yet in the product, namely partition tuning, histogram-guided recursion, and caching results of expensive functions using hybrid caching [Hellerstein 1996]. Similar to the last feature, we implemented duplicate removal with fast return of the first result rows, which uses the same control flow, and are missing only a few mechanisms for record formats and function invocation. Beyond these techniques, we hope to include several further improvements in a later release. The most prominent of those is dynamic memory adjustment during the run-time of a join, including restoring spilled partitions dynamically in order to exploit memory made available in the middle of a partitioning step.

Memory management

Goals

Previous releases have relied very heavily on loops- and index-based methods for set operations such as joins and grouping. For any one query, at most one sort operation could be active at a time. Thus, division of memory between queries (as well as other types of requests, such as index creation) as well as within queries was a lot simpler than the current version, which uses query plans with multiple concurrent sorts (e.g., two sorts feeding into a merge join that in turn feeds a third sort operation), multiple concurrent hash operations (within and between teams), as well as mixed plan with sort and hash operations active concurrently, feeding data to each other.

For simplicity and robustness, we decided to forgo potential benefits of dynamic memory adjustment during a query's (or a request's) run-time and instead run each query with an amount of memory set at start-up time. There are three remaining problems that need to be solved. First, each query needs to be admitted for processing. Second, an amount of memory must be set for the query. Third, memory must be managed among the operators within a query plan.

In this paper, we do not detail our final solution for the first and second problems. A simple and robust approach is to set aside a fraction of the buffer pool for query memory, say 50%, to assign each query with at least one memory-intensive operation a fixed amount of memory, say 1 MB or 4 MB, and to admit additional queries until the entire query memory is assigned to running queries. A slightly more sophisticated policy also considers CPU load.

The third problem is still quite hard. Our goal was to create a solution that permits fairly efficient execution of complex plan trees yet is complete, robust, and simple. These latter three issues are imperative in a product environment, even if the policy incurs at times some minor performance loss. One criterion for robustness is that the policy degrades gracefully if the optimizer's selectivity estimates are wrong, an issue that has often been ignored in the literature on resource allocation.

Rejected solutions

We considered a variety of alternative solutions for each of the problems, but decided to use policies that are simple and robust rather than optimal but complex. A possible policy for dividing memory between the general I/O buffer and query memory is based on *turn-over rates* and was inspired by the "five-minute rule" [Gray 1987]. Both the I/O buffer and the memory-intensive operators employ main memory to hold disk pages and therefore reduce the amount of I/O. The basic idea of buffer replacement policies is to retain those pages in memory that are used frequently. The basic idea of the allocation policy considered here is to ensure that there is a uniform cutoff frequency for all pages deemed worthy to be retained in memory. For pages in the general I/O buffer, this frequency is fairly easy to determine. For pages in the system-wide query memory, the turn-over frequency is based on the merge behavior of external sorting or the partitioning behavior of hash-based algorithms with overflow, and on the total I/O bandwidth of devices used for run files in sorting and overflow files in hash algorithms. We presume that all temporary files are striped to balance the I/O load among these devices, which is generally a simply and effective policy. Note that concurrent sort or hash operations multiplex both the query memory and these I/O devices with compensating effect (for the consideration here), so we

ignore concurrent operations. Given that both merge and hash algorithms need to read and write temporary files, data pass through the query memory at half the bandwidth of these I/O devices. The quotient of half this I/O bandwidth and the size of the query memory is the turn-over frequency in the query memory. The proposed policy is to adjust the division of memory between general I/O buffer and query memory such that the turn-over rate or frequency in the general I/O buffer and in the query memory are equal.

A possible policy for dividing memory between competing queries is to compute upper and lower memory limits for each query with at least one memory-intensive operator. A useful upper limit is the maximal useful memory, i.e., the amount of memory with which, based on the anticipated sizes of intermediate results, the entire query plan can be executed in memory, without any hash overflow files or sort run files being written to disk. Moreover, the upper limit should be restricted not to exceed the server's query memory. The lower limit must be at least large enough to permit reasonably effective merging in external sort as well as partitioning in hash operations, which may translate to a fan-in and fan-out of at least 8 or 12 in intermediate phases of sorting and hash matching. If there is memory contention, assign all concurrent queries the same fraction of their maximal useful memory, but no less than their minimum memory.

Policy

Our solution for managing memory within a query plan first divides a complex plan in phases based on the input, intermediate, and output phases of all stop-and-go operators in the plan and then, still at compile-time and therefore based on expected cardinalities and average record lengths, assigns to each operator phase a fraction of the memory available to the query as a whole. Intermediate phases are always assigned 100% of the available memory. When multiple operators make up a plan phase, memory is assigned to each operator proportionally to the operator's anticipated input data volume relative to the total input data volume of all memory-intensive operators participating in the plan phase. For hash joins, it is presumed that hybrid hashing is used, i.e., the hash join's second phase competes with both the probe input and the output. The memory assignment for a hash join is the minimum assignment among its input phases. A team of hash iterators is treated as a single N-ary hash join with many phases.

At run-time, each operator phase is permitted to allocate up to its fraction of the available memory. If an iterator holds memory, the iterator and its memory are said to be *waiting* if the iterator has completed its *open* method and is ready for the first invocation of its *next* method. Similarly, an iterator and its memory are waiting if the iterator has invoked an *open* method on one of its inputs which has not returned yet. For example, if a merge join is surrounded by three sort iterators, both the sort iterator at the root of the plan segment and the sort iterator for the first input are waiting while the merge join opens its second input. As a second example, a hash join and its memory are waiting while the probe input is being opened.

While an operator is waiting, its memory is registered with the query memory manager. If some other operator in the query needs to allocate memory, the query memory manager requests that one of the registered waiting operator release some or all of its memory. The operator chosen is the operator registered (and waiting) the longest, because that operator is probably the furthest away within the query plan from the plan phase currently active. In other words, memory-intensive operators that may benefit from holding data in memory between their *open* method and the first invocation of their *next* method must be able to release some or all of this memory upon request. For example, if the first input into a merge join is a sort with a data volume small enough to fit in memory (and therefore kept in memory between the sort iterator's *open* and *next* methods), and if the merge join's second input is a sort iterator that needs all available memory for an intermediate merge, the query memory manager will request that the first sort spill its memory contents to disk, in a sense turning the in-memory sort into an external sort with a single run file. The important issue is that this is done on demand only, because the optimizer's size estimates for the second sort's input may have been wrong in either direction. Similarly, a hash join in a complex plan, e.g., a bushy plan, must be prepared to spill and later restore its hash table if the probe input is itself a complex plan that requires a lot of memory. Fortunately, restoring some or even all partitions after the probe input has been opened, but before it has been consumed, is one of the operations inherent in partition tuning, and thus needs to be implemented anyway.

Performance

For our performance study, we chose the TPC-D verification database with 100 MB of raw data (scale factor 0.1) and a desktop PC running Windows NT 4.0. In order to force memory contention, each query is limited to 1 MB of memory for sort and hash operations. The buffer is flushed before each query plan is started. Because our product's latest release that includes all the described techniques is still under development, we cannot report absolute "wall clock" times. Given the incessant improvements in processing and I/O hardware, relative times serve our purpose just as well.

The *LineItem* table contains about 600,000 rows of about 120 bytes (about 72 MB), and the *Orders* table holds 150,000 rows of about 100 bytes (about 15 MB). Our physical database design is fairly simple but not untypical: a clustered index on each table's primary key, a non-clustered index on each foreign key, and non-clustered indexes on some further columns, e.g., *Orders.O_OrderPriority* and *LineItem.L_ShipDate*. Note that if a clustered index exists for a table, our storage engine requires that non-clustered indexes for that table contain the search key of the clustered index, similar to Tandem's database products.

Rather than reporting on the performance of the original TPC-D query set, which tests the query optimizer as much as the execution algorithms, we chose two simple and typical queries to show interesting alternative plans. Typically, our optimizer selects the plan with the lowest anticipated cost, but specific plans can also be forced using hints.

Single table query

The first example query retrieves and counts rows from the *Orders* table: *Select O_OrderDate, O_OrderPriority, Count (*) From Orders Where O_OrderDate between '1994/1/1' And '1994/3/31' Group By O_OrderDate, O_OrderPriority.* This query summarizes about 3.8% of the *Orders* table or 5,700 orders. The optimizer cannot find a single covering index for this query; therefore, it considers three plans for retrieving data from the *Orders* table. These plans scan the base file, scan a non-clustered index on *O_OrderDate* with subsequent record fetch, or join two indexes as described in the introduction. Given that the two indexes are not sorted on row identifiers, hash join is used in the third plan. Note that the index on *O_OrderPriority* is scanned in its entirety, all 150,000 index entries, whereas only 3.8% of the index on *O_OrderDate* is scanned. The grouping operation is performed by hash grouping in all three plans. Neither the hash join nor the hash grouping spill overflow files to disk.

Plan	CPU time	Elapsed time
Scan the base table	81.25	100.00
Scan index O_OrderDate and fetch from base table	38.25	789.85
Join indexes O_OrderDate and O_OrderPriority	70.35	73.45

Table 1 – Obtaining records from one table.

The run times of these three plans are shown in Table 1, scaled to compare with the elapsed time of a base file scan, which is the optimal available plan for this query in most database systems. While fetching guided by a single index consumes very little CPU time, it results in an excessive elapsed time due to the large number of random disk reads. The fastest plan for this query is to join the two indexes on their shared row identifier. This plan incurs both less CPU time and less elapsed time than scanning the base file. For this very simple query representing a very well known and understood type of query, this plan achieves savings of 25% over the best plan available in most database systems. Note that joining two indexes, in effect a form of vertical partitioning, has been shown viable here even for tables with records as short as 100 bytes. For larger records in the base table, the benefit increases.

Join query

The second example query joins two large tables and computes a grouped summary: *Select O_OrderKey, O_OrderDate, Count (*) From Orders, LineItem Where O_OrderKey = L_OrderKey And L_ShipDate ≥ '1994/1/1' Group By O_OrderKey, O_OrderDate.* The date restriction is satisfied by about 72.5% of the *LineItem* rows, or 435,000 line items. Note that the indexes on *Orders.O_OrderDate* and *LineItem.L_ShipDate* are covering indexes for this query. Therefore, the traditional execution plan is to scan these two indexes, to join the results on *OrderKey*, and then to group the join result on *O_OrderKey, O_OrderDate.* In many database systems, the optimizer will choose this plan. However, our optimizer realizes that *O_OrderDate* is functionally dependent on *O_OrderKey* and removes the dependent column from the grouping list. Thus, the join and the grouping operations hash on the same set of columns (the singleton set *O_OrderKey*), and the two hash operations can be executed as a team. Moreover, the grouping operation can be pushed down through the join operation. Finally, since our hash join operation supports grouping or duplicate removal on the build input, the optimizer may choose the opposite-to-normal roles of *Orders* and *LineItem* in the hash join. Doing so permits grouping of *LineItem* records and joining with *Orders* records using a single binary operator with a single hash table per partitioning step.

Last operation	Hash team	Build or outer input	CPU time	Elapsed time
Grouping	No	Orders	30.47	100.00
Grouping	No	Line items	33.38	101.86
Grouping	Yes	Orders	19.14	59.62
Grouping	Yes	Line items	19.73	59.19
Join	No	Orders	17.74	63.88
Join	No	Line items	17.68	62.29
Join	Yes	Orders	15.62	51.46
Join	Yes	Line items	13.59	46.44
Integrated	N/A	Line items	12.46	48.16

Table 2 – Join and grouping from two tables.

Table 2 shows the performance of nine plans. Times are relative to the elapsed time of the traditional plan, which is indicated in the first row. All nine plans scanning the two covering indexes and using hash join and hash grouping; therefore, performance differences are due to the join and grouping strategies, not due to different scans or other plan differences. Comparing the first two rows, it is interesting to note that the choice of build inputs for the join hardly affects the performance, in spite of the fact that the *LineItem* table is significantly larger than the *Orders* table, even after the selection on

L_ShipDate. The reason is role reversal, which ensures that the smaller overflow file is used as build input during each overflow resolution step. Moreover, bit vector filtering reduces the fraction of *Orders* rows that are written to overflow files, in effect transferring the reduction of the *LineItem* table to the *Orders* table before their join is even complete.

In a comparison of the first two rows with the subsequent two rows, it becomes obvious that organizing multiple hash operations into a team can substantially improve performance. A team operation saves all or most overflow I/O for intermediate results within the team. In this query, the join result is about as large as the two join inputs together. Thus, saving the I/O for the join result substantially improves performance. For the example query, teams reduce both CPU time and elapsed time by about 40%.

The next four rows reflect the optimizer's ability to invert the sequence of join and grouping operations. Performance is consistently better than plans using the traditional processing sequence. Note that designating the *LineItem* table as build input is now advantageous, because the result of the grouping operation is smaller than the *Orders* table. Again, hash teams are very effective. Their relative effect of teams is reduced to about 20%, because overflow I/O is saved for only one of the two join inputs. However, 20% is still substantial, given that this is a fairly simple and well-studied type of query.

The final row indicates the performance of integrating grouping on the build input with the join. While CPU consumption is lower than in the most similar plan (the row above), elapsed time has increased. We suspect that the reason is that role reversal is inhibited in the integrated algorithm, and that bit vector filtering and role reversal combine in the team plan to reduce the total I/O volume and elapsed time. The important result of this experiment is that for this very typical query, hash teams or the integrated operation are required to improve the elapsed time to less than half of the elapsed time of the traditional plan.

Summary and conclusions

In this paper, we have described how we have combined many of the hashing techniques proposed in the research literature into a single operator. Based on the reported experiments using the TPC-D database with 100 MB of raw data, as well as many other experiments not reported here, we believe that the performance of our algorithms is very competitive.

The hash operation described in this paper is novel in two aspects. First, it is the first implementation of N-ary hashing or teams of hash operations. Second, it cleanly integrates into a single, reasonably clean and extensible implementation a wide array of advanced hashing techniques proposed and prototyped individually by various research groups. The substantial performance gains for two very simple, very well studied, and very typical "work horse" queries clearly demonstrate that this integration as well as teams are truly worthwhile, yet that no single one of the techniques is a panacea for high performance hashing. In other words, for optimal performance, teams must be integrated with all the other hashing techniques.

The memory management technique described in this paper is most notable for being simple yet very effective. It adapts to all types of query plans, including complex bushy plans with multiple sort and hash operations. It exploits the execution model based on iterator objects with *open*, *next*, and *close* methods, divides complex execution plans into plan phases based on operator phases in stop-and-go operators, keeps track of active and waiting operations, and mimics LRU among all active and waiting operators. It is simple, probably not always optimal, but very robust and on the whole very effective.

References

Bratbergsengen 1984: Kjell Bratbergsengen: Hashing Methods and Relational Algebra Operations. VLDB Conf. 1984: 323-333.

DeWitt 1984: David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, David A. Wood: Implementation Techniques for Main Memory Database Systems. ACM SIGMOD Conf. 1984: 1-8.

DeWitt 1985: David J. DeWitt, Robert H. Gerber: Multiprocessor Hash-Based Join Algorithms. VLDB Conf. 1985: 151-164.

DeWitt 1993: David J. DeWitt, Jeffrey F. Naughton, J. Burger: Nested Loops Revisited. PDIS Conf. 1993: 230-242.

Fushimi 1986: Shinya Fushimi, Masaru Kitsuregawa, Hidehiko Tanaka: An Overview of The System Software of A Parallel Relational Database Machine GRACE. VLDB Conf. 1986: 209-219.

Graefe 1993: Goetz Graefe: Query Evaluation Techniques for Large Databases. ACM Computing Surveys 25(2): 73-170 (1993).

Graefe 1994: Goetz Graefe: Sort-Merge-Join: An Idea whose Time Has(h) Passed? Data Eng. Conf. 1994: 406-417.

Gray 1987: Jim Gray, Gianfranco R. Putzolu: The 5 Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. ACM SIGMOD Conference 1987: 395-398.

Hellerstein 1996: Joseph M. Hellerstein, Jeffrey F. Naughton: Query Execution Techniques for Caching Expensive Methods. ACM SIGMOD Conf. 1996: 423-434.

Kitsuregawa 1989: Masaru Kitsuregawa, Masaya Nakayama, Mikio Takagi: The Effect of Bucket Size Tuning in the Dynamic Hybrid GRACE Hash Join Method. VLDB Conf. 1989: 257-266.

Red Brick 1996: Company Press Release: Red Brick Launches Red Brick Warehouse 5.0 for Data Warehouse, Data Mart, Data Mining, Database Marketing and OLAP Applications, Los Gatos, CA, Oct. 14, 1996; see www.redbrick.com.

Sacco 1986: Giovanni Maria Sacco: Fragmentation: A Technique for Efficient Query Processing. ACM TODS 11(2): 113-133 (1986).

Schneider 1990: Donovan A. Schneider, David J. DeWitt: Tradeoffs in Processing Complex Join Queries via Hashing in Multiprocessor Database Machines. VLDB Conf. 1990: 469-480.

Sedgewick 1984: Robert Sedgewick, Algorithms, Addison-Wesley 1984.

Selinger 1979: Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price: Access Path Selection in a Relational Database Management System. ACM SIGMOD Conf. 1979: 23-34.

Zeller 1990: Hansjörg Zeller, Jim Gray: An Adaptive Hash Join Algorithm for Multiuser Environments. VLDB Conf. 1990: 186-197.