Executing Nested Queries

Goetz Graefe
Microsoft Corporation
Redmond, WA 98052-6399

Abstract: *Optimization* of nested queries, in particular finding equivalent "flattened" queries for queries that employ the SQL sub-query construct, has been researched extensively. In contrast, with the exception of nested loops join, *execution* of nested plans has found little interest. Nested execution plans may result from a failure to flatten nested SQL expressions but just as likely are created by a query optimizer to exploit all available indexes as effectively as possible. In fact, if materialized views and index tuning perform as expected, few queries should require large operations such as parallel scans, sorts and hash joins, and most actual query plans will rely entirely on navigating indexes on tables and views. Note that only index navigation plans scale truly gracefully, i.e., perform equally well on large and on very large databases, whereas sorting and hashing scale at best linearly. Since a typical index navigation plan employs nested iteration, this paper describes techniques to execute such plans efficiently as well as means to cleanly implement these techniques. Taken together, these techniques can improve query performance by orders of magnitude, giving them obvious practical importance.

1 Introduction

Database and disk sizes continue to grow fast, whereas disk access performance and disk bandwidth improve much more slowly. If for no other reason than that, research into database query processing must refocus on algorithms that grow linearly not with the database size but with the query result size. These algorithms and query execution plans rely very heavily on index navigation, i.e., they start with a constant given in the query predicate, find some records in an index, extract further column values from those records, find more records in another index, and so on. The cost of this type of query plan grows linearly with the number of records involved, which might very well mean that the cost is effectively independent of the database size. Actually, the cost of index lookups in traditional B-tree indexes grows logarithmically with the database size, meaning the cost doubles as a table grows from 1,000 to 1,000,000 records, and doubles again from 1,000,000 to 1,000,000,000,000 records. The cost barely changes from 1,000,000 to 2,000,000 records, whereas the cost of sort or hash operations doubles. Moreover, it is well known that scanning "cleans" the I/O buffer of all useful pages, unless the replacement policy is programmed to recognize scans [S 81]. It is not likely, however, that CPU caches and their replacement policy recognize scans; thus, large scans will repeatedly clear out all CPU caches, even level-2 and level-3 caches of multiple megabytes. Based on these behaviors, on the growth rates of disk sizes and disk bandwidths, and on the recent addition of materialized and indexed views to mainstream relational database systems, we should expect a strong resurgence of index-based query execution and thus research interest in execution plans relying heavily on nested iteration.

Figure 1 shows a query evaluation plan for a simple join of three tables, with the "outer" input of nested iterations shown as the left input. The nested iterations are simple Cartesian products here, as the join predicates have been pushed down into the inner query plans. If the filter operations are actually index searches and if table T0 is small relative to T1 and T2, this plan can be much more efficient than any plan using merge join or hash join. An alternative plan uses multiple branches of nested iteration rather than multiple levels as the plan in Figure 1; this plan would be left-deep rather than right-deep. Of course, complex plans might combine multiple levels and multiple branches at any level.

Nested loops join is the simplest form of nested iteration. There is no limit to the complexity of the inner query. It might require searching a non-clustered index and subsequently fetching from the main table; intersecting two or more non-clustered indexes using bitmap, sort, or hash operations; joining multiple tables each with its own access paths; multiple levels of nested iteration with multiple sub-branches at each level; etc. For all but the most trivial cases, there are complex issues that need to be addressed in a commercial product and in a complete research prototype. These issues exist at the levels of both policies and mechanisms, with avoiding I/O or exploiting asynchronous I/O as well as with managing resources such as memory and threads.

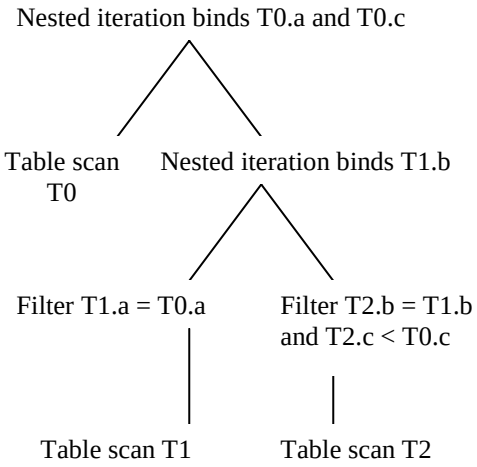

Figure 1. A query execution plan with moderately complex nested iteration.

The purpose of this paper is to explore these issues, to describe proven solutions and promising approaches, to detail how to implement those solutions, and to focus and stimulate research into open issues. Many useful techniques have long been known and are surveyed here, some for the first time. Other approaches and solutions are original contributions, e.g., the use of merged indexes for caching in nested iteration or how to manage batched correlation bindings in the outer and inner plans. For all of those techniques, data flow and control flow are detailed, including new iterator methods for clean and extensible implementations of the described techniques in sequential and parallel execution plans. The following section sets the stage for the remaining ones, which cover, in this order, asynchronous I/O, avoiding I/O through caching and other means, data flow, and control flow.

2 Background, scope, and assumptions

In the following discussions, we do not consider query optimization, e.g., "flattening" or "unnesting" or "decorrelating" nested query expressions into ones that do not employ nesting. While an interesting and traditionally challenging topic of research and commercial development, we consider it a complement to the present paper. Similarly, we presume here that the physical database design is given and cannot be tuned. Again, this is an important and challenging complementary topic in itself. We also do not describe algorithms or data structures for indexes, as those issues are covered elsewhere.

We presume typical contemporary database server environments, i.e., a large number of user connections and concurrent transactions; a small or moderate number of CPUs, each with its own cache; a fair amount of main memory divided among many connections, queries, operators, and threads; a number of disks larger than the number of CPUs by a small factor, say ten times more disk drives than CPUs; a database on disk many times larger than main memory; some virtualization in the disk layer such as caches in disk controllers, system-area networks, and network-attached storage; many tables of varying sizes, each with a small number of B-tree indexes on a few columns; non-clustered indexes pointing into heap files using record identifiers (page number and slot number) or into clustered indexes using unique search keys; etc. Without doubt, some of the techniques described here need to be adjusted if the environment changes drastically, e.g., if the entire database resides in main memory or if all indexes use hashing rather than B-trees.

For the sake of completeness, we very briefly review the various forms of nested loops join. *Naïve nested loops join* performs a simple scan of the inner input for each row of the outer input. If the outer and inner inputs are stored tables, *block nested loops join* actually employs four nested loops: the two outer-most loops iterate over pages (or blocks of some size) of the outer and inner inputs, and the two inner-most loops perform naïve nested loops among the rows in a pair of blocks. I/O volume is reduced by the blocking factor of the outer, i.e., the number of records in a block of the outer input, but the join predicate is evaluated just as often as in naïve nested loops. I/O will be slightly reduced if scans of the inner input go forward and backward, because of buffer effects each time the direction is reversed [K 80].

However, unless the inner input is very small, e.g., less than a few dozen rows, the only version of nested loops join used in practice is index nested loops join. Many query optimizers consider temporary indexes for the inner input, if no useful permanent index is available. These indexes are used only during a single execution of a specific query plan, and their contents are usually more restrictive than an entire stored table. The type of index does not really matter – it might be a B-tree index, a hash index, a bitmap index, a multi-dimensional index, etc. – as long as the index supports the join predicate efficiently and effectively, i.e., without fetching many more index entries than truly necessary.

In general, nested iteration works better than set-oriented algorithms such as hash join if the outer input is very small relative to the database, such that navigating the inner input using indexes leaves most of the database untouched. Typical queries include not only OLTP requests but also cursor operations such as "fetch N more result rows" and their internet-age equivalents for "next page" queries with their typical "top" operations. In all

these cursor-like operations, resuming where the prior request left off requires sort order, and nested iteration preserves the sort order from the outer input, and it might even extend it with additional ordering columns from the inner input.

A special case of a small outer input is a set of parameters left unbound in the query text, to be bound when an application invokes the query. This situation can be modeled, both in optimization and in execution, as a nested query. The outer input is a single row with a column for each parameter, and the inner input is the entire original query. This is one of the few cases in which a query optimizer can truly rely on its "estimate" for the size of the outer input (other cases include exact-match predicates on unique columns and range selections on unique integer columns). If a parameterized query is optimized and executed as a nested query, it is easy to see how "array invocations" with multiple sets of parameters can be supported.

Even for large queries, nested iteration can be more efficient than merge joins or hash joins, in particular if covering indexes are available, i.e., indexes that deliver all columns required in a given query without fetching rows from the base table. Consider a self-join of a table *order details* on the (non-unique) column *order key*, e.g., for an analysis of purchasing combinations. A hash join needs to scan the entire table twice, with no beneficial buffer effects if the table size exceeds the buffer size, and it requires a massive amount of memory as well as additional I/O for overflow files. In a merge join of two pre-sorted inputs, the two scans benefit from sharing pages in the I/O buffer, but the merge join requires support for the backup logic of many-to-many merge joins, often implemented by copying all records from the inner input into the local buffer. An index nested loops join, however, benefits from the I/O buffer such that all data pages need to be read only once, and it exploits the index on *order key* to find matching inner rows for each outer row. In general, in a well indexed database, the disk access patterns of index nested loops join and merge join are often very similar [GLS 93], and index nested loops join performs as well as or better than merge join.

DeWitt et al. [DNB 93] found that index nested loops join can best other joins, even on parallel database servers considered by many showcases for hash-based join algorithms. The essence is that I/O cost often dominates join cost, and that index nested loops performs well if the index on the inner input can ensure that only a fraction of the inner input must be read from disk. In fact, it has to be a small fraction, since sequential I/O (as used for hash join) is much more efficient than random I/O (as used by index nested loops join), in particular if asynchronous read-ahead overlaps processing time and CPU time.

3 Asynchronous I/O

Asynchronous I/O is, however, not limited to sequential scans. In a naïve implementation of index nested loops, each index lookup and record fetch is completed before the next one is initiated. Thus, there must be at least as many threads as disks in the system, because otherwise some disks would always be idle. This would not be good given that disk I/O is a precious resource in database query processing.

Index nested loops join can, however, if implemented correctly, exploit asynchronous I/O very effectively. In some recent IBM products, unresolved record identifiers are gathered into fixed-sized lists, and I/O is started concurrently for all elements in the list once the

list is filled up. Some recent Microsoft products keep the number of unresolved record identifiers constant – whenever one record has been fetched, another unresolved record identifier is hinted to the I/O subsystem. In general, steady activity is more efficient than bursts of activity.

Fetch operations can exploit asynchronous I/O for heap files, when unresolved record identifiers include page numbers, as well as for B-tree indexes, where unresolved references are actually search keys. There are two effective methods for combining synchronous and asynchronous I/O in those situations, both relying on a *hint* preceding an *absolute* request for the desired page. First, while descending the B-tree searching a hinted key, the first buffer fault can issue an asynchronous read request and stop processing the hint, which will be resolved later using synchronous I/O for each remaining buffer fault. Second, presuming that in an active index practically all pages above the leaf pages will reside in the buffer, searching for a hinted key always uses synchronous I/O for the very few buffer faults for nodes above leaves, and always issues an asynchronous I/O request for the leaf page. In practice, the two methods probably perform very similarly.

The generalization of such asynchronous I/O for complex objects was described and evaluated by Keller et al. and Blakeley et al. [KGM 91, BMG 93], but not adopted in any production system, to the best of our knowledge. The central idea was to buffer a set of object roots or root components such that the I/O system always had a substantial set of unresolved references. These would be resolved using a priority queue, thus attempting to optimize seek operations on disk. If object references are logical, i.e., they require mapping to a physical address using a lookup table, both lookups should use asynchronous I/O.

Always keeping multiple concurrent I/Os active can be a huge win. Most server machines have many more disk arms than CPUs, often by an order of magnitude. Thus, keeping all disk arms active at all times can improve the performance of index nested loops join by an order of magnitude. Even in a multi-user system, where many concurrent users might keep all resources utilized, reduced response times lead to higher system throughput due to reduced lock contention. Keeping many disks busy for a single query requires either aggressive asynchronous I/O or many threads waiting for synchronous I/O. For example, in some Informix (now IBM) products, the number of parallel threads used to be determined by the number of disks, for precisely this reason. It is probably more efficient to limit the number of threads by the number of CPUs and to employ asynchronous I/O than to use very many threads and only synchronous I/O, since many threads can lead to excessive context switching in the CPUs, and more importantly in the CPUs' data caches. This presumes, of course, that I/O completion is signaled and not actively and wastefully polled for.

4 Avoiding I/O

As attractive as it is to speed up I/O, it is even better to avoid it altogether. The obvious means to do so is caching, e.g., in the file system's or the database manager's buffer. When computing complex nested queries, however, it is often beneficial to cache results even if they may not remain in the I/O buffer and may require I/O, because I/O for the cache might be substantially less than I/O to recompute the cache contents. Note that caching avoids not only repetitive computation but also repetitive invocations of the lock

manager, and that the I/O buffer usually can retain cached results more densely than the underlying persistent data. Caching has been used in several products, e.g., IBM's DB2 for MVS [GLS 93] as well as recent Microsoft database products.

It is well known that the result of an inner query can be cached and reused if the inner query does not have any references to outer correlation values, i.e., the inner query is equivalent to a constant function. Since this is the easy case with a fairly obvious implementation, we do not discuss it further here. Similarly, in a nested iteration where the outer input is guaranteed to have only a single row, caching is not very difficult or interesting (at least not caching within a single query).

Expensive user-defined functions are closely related to nested queries; in fact, since either can invoke the other, it is a chicken-and-egg problem to decide which one is a generalization of the other. In research and in practical discussions, there is usually a difference in emphasis, however. First, functions are usually presumed to invoke user-defined code with unpredictable execution costs, whereas nested queries usually perform database searches, where standard selectivity estimation and cost calculation apply, with buffer (caching) effects as added complexity. Second, functions usually are presumed to have scalar results, i.e., a single value, whereas nested queries often produce sets of rows.

4.1 Caching results of the inner query

For a cache, any associative structure such as a hash table or a B-tree index at the root of an inner query plan is sufficient. The search key is the correlation value, and the retrieved information is the result of the inner query for that correlation value. A single simple lookup structure permits caching scalar values, e.g., a sum, an average, or a Boolean flag whether or not the result of the inner query is empty or unique, i.e., free of duplicate rows [HN 96]. If the result is a set of rows, it might be necessary to employ two lookup structures, one index to indicate whether a correlation value has occurred before and another one with the actual results. If only one index is used, it is not clear how to remember that some correlation value produced an empty set. The first index, which we may call the control index, could be a bitmap index, but it could not be a standard bitmap filter, since bitmap filters usually permit hash collisions. It could be restricted to contain only those outer correlation values that resulted in an empty inner result, or it could contain all outer correlation values already seen.

If the outer input binds multiple correlation columns as parameters of the inner query, the search key of the cache structure contains multiple search columns. If different branches of the inner query's execution plan use different subsets of these correlation values, it is also possible to cache not at the root of the inner query but at the root of those branches. The advantage is that each of the branches is executed fewer times; the disadvantage is that the computation between the roots of the branches and the entire inner query is executed multiple times. In extreme cases, it is possible to cache in multiple places, but considering too many combinations of caches can result in excessively complex and expensive query optimization.

For example, consider the query execution plan in Figure 2, which differs from Figure 1 as there is only one nested iteration and a join or intersection within the inner plan. Note that tables T1 and T2 could also be different indexes of the same table in a plan using

index intersection for a conjunctive search predicate. Point A is the root of the inner plan and a good place to cache inner results. However, it might also be possible to cache at points B or C, or even at all three points. If T0 contains 10,000 rows, T1 and T2 are scanned 10,000 times each, unless caching is employed. If T0 contains only 1,000 distinct combinations of the pair (a, b), caching only at point A means that T1 and T2 are scanned 1,000 times each. If, on the other hand, T0 contains only 100 distinct values of (a) and only 200 values of (b), caching at points B and C means that T1 and T2 are scanned only 100 and 200 times, respectively, although the intersection operation is still performed 10,000 times. Caching at all three points ensures both the lowest counts of intersection operations (1,000) and of scans for T1 and T2 (100 and 200).

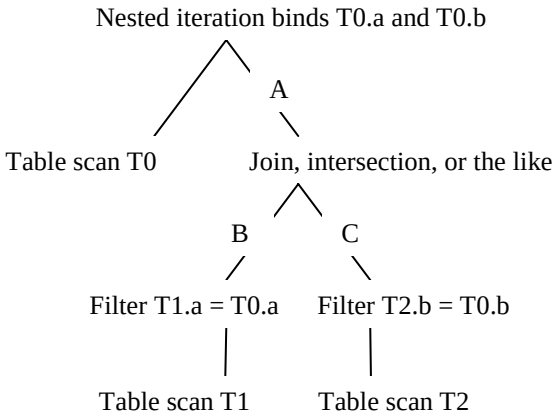

Figure 2. A query execution plan with moderately complex nested iteration.

Many query optimizers will consider the costs and benefits of caching only at point A for a query of this type. Otherwise, the optimization time becomes excessive – even for this simple example, there are 2^3 alternative caching strategies to consider. Moreover, different rows in T0 with different values in (a) and (b) may result in different selectivities in the filters on T1 and T2, and the chosen caching strategy must work well for all of those. Fortunately, in most cases, the inner query plan relies on index searches rather than table scans, and indexes can create powerful cache effects of their own.

For example, if T1 is searched with an index on (a), the index itself serves the role of a cache at point B. Moreover, one can sort the outer input on the correlation values, such that equal values are in immediate succession, and inner query results can be reused instantly. By sorting the outer input T0 on (a) or by scanning an index on T0.a, the index searches for T1 will be very "orderly," creating a disk access pattern quite similar to a merge join of T0 and T1. If multiple rows from a single leaf page within the index on T1.a participate in the query result, it is guaranteed that this leaf will be fetched only once, independent of the size of the buffer pool and its replacement policy. However, if there are multiple correlation columns as in this example, it is not clear on which column sequence to sort. In the example above, should the optimizer elect to sort the outer input T0 on (a), on (a, b), on (b), or on (b, a)? If input T0 has an index only on (a) but not on (a, b), is an explicit sort operation worthwhile? Quite obviously, only a cost-based query optimizer can answer such questions, case by case.

There is a related question about sorting the result of searching an index within the inner query. If there is a useful index, say on column (a) of table T1, how is the result cached? If the join operation between T1 and T2 uses a predicate on column (c) in both T1 and T2, the join can be computed using a fast merge join without sorting if, for any value of the pair (a, b) from T0, the caches at points B and C retain values sorted on (a, c) and (b, c), respectively. In other words, the sort on (c) is part of the computation cached at points B and C, enforced prior to caching intermediate results and not repeated when retrieving data from the caches. Using the specific counters assumed earlier, the merge join might be executed 1,000 or 10,000 times depending on caching at point A, but there will be only 100 sort operations for rows from T1 and 200 sort operations for rows from T2. In this case, the benefit of the caches at B and C could be savings in sort effort, whether or not there are worthwhile savings in I/O for the permanent data.

Unfortunately, if a cache grows large, it may cause I/O cost of its own. In that case, it is possible to limit the total size of a cache by subjecting its entries to a replacement policy. Units of insertion and deletion from the cache are correlation values from the outer input of the nested iteration. Promising replacement policies are LRU (least recently used) and LFU (least frequently used); a reasonable measure of "recent" can be a sequence number attached to the outer input (using a "rank" operation, if available). Both policies can be augmented with the size of an entry, i.e., the size of an inner result for an outer row, and the cost of obtaining it, if this cost differs among outer rows. If the cache is represented in two indexes, one containing only previously encountered outer correlation values and one containing result sets from the inner query, information on costs and usage belong into the first one. The replacement policies for the two indexes do not necessarily have to be the same, such that statistics might be kept even for outer correlation values that have temporarily been evicted from the cache of inner results – reminiscent of the LRU-k policy that has been proposed for disk buffers [OOW 93].

Materialized and indexed views can be exploited as special forms of the caches discussed so far. If incremental maintenance of materialized views with join and semi-join operations is supported, permanent auxiliary tables can perform the role of control indexes discussed above. For any value currently found in the control table, the materialized view contains the result of the nested query. Creation of materialized views and their control tables can be left to an index tuning tool or perhaps even the query optimizer. Query execution can simply ensure that the actual parameter values are present in the control table and then immediately retrieve the results rows from the result table. "Ensuring" presence of certain rows in the control table uses the "merge" operation of ANSI SQL, sometimes also called "insert or update," and its implementation in query execution plans. If control rows contain information beyond the actual parameter values, e.g., last-used time stamp for use by an LRU replacement scheme, there is indeed an update component. The transactional context for inserting a new row into the control table should be a system transaction, not the user transaction and its isolation level etc., although this system transaction should run within the user's thread (similar to a system transaction that splits a B-tree page). The user transaction then locks the control rows it reads, precisely the same way as rows in an ordinary table or in a materialized view. An asynchronous system task can remove rows from the control table that are not worth keeping, which of course instantly removes the appropriate result rows from the materialized view and all its indexes.

4.2 Implementation of sorting

Sorting the outer input is worthwhile with respect to I/O in the inner plan if it can achieve advantageous buffer effects for page accesses in the inner plan. If a table or index accessed in the inner plan is so small that it will fit entirely in memory or in the I/O buffer, sorting the outer does not reduce I/O in the inner. Conversely, if the table or index accessed in the inner is so large that there are more pages (or whatever the unit of I/O is) than there are rows in the outer input, sorting the outer does not reduce I/O in the inner, except if the outer input contains rows with duplicate correlation values.

In order to achieve the benefits of sorting with the least run-time cost, the implementation of sorting must be integrated neatly with the query processor. Intermediate result rows on the input of the sort operation must be pipelined from the prior query operation directly into the sort, and sorted rows must be pipelined directly into the next operation. In other words, the only I/O due to a sort operation should be sorted intermediate run files. A sort implementation that is limited to reading its input from a stored table or index and writing its output to a stored table or index may triple the I/O cost due to the sort. Nonetheless, despite just such an implementation of sorting, DeWitt et al. observed substantial performance advantages for nested loops join due to sorting the outer input [DNB 93].

Rather than sorting the outer input completely, sorting only opportunistically might achieve sufficient locality in the inner input to realize most of the potential savings. For example, if the cost of an external merge sort is high, the optimizer might choose to run the outer input through the run generation part of an external merge sort but without writing any runs to disk. This technique is promising if the number of records in the generated runs is larger than the number of pages in the file from which matching records are fetched. In addition, if the inner input is stored in a clustered index, sorting the outer might improve cache locality while searching the clustered index. In order to preserve a steady flow of intermediate result records within the query plan, replacement selection based on a priority queue is probably a better algorithm than read-sort-write cycles typically associated with quicksort. Moreover, replacement selection on average yields runs twice as long as quicksort.

4.3 Merged indexes

Another means to avoid I/O is physical clustering. Master-detail clustering in relational and other database systems has long been known to reduce I/O when nested iteration plans assemble complex objects [H 78].

As a prototypical example, consider how to merge the clustered indexes for the relational tables *customers*, *orders*, *order details*, *invoices*, and *invoice details*. The goal is that all information about a single customer is in a single leaf page or in adjacent leaf pages. The significant characteristics of the example are that it includes both multiple levels and multiple branches at one level, and that it mixes “strong entities” and “weak entities”, i.e., ones with their own unique identifier, e.g., *order number*, and ones which rely for their existence and identity on other entities, e.g., *line number* within an *order*.

One very useful way to think about master-detail clustering is as multiple traditional single-table indexes merged into a single B-tree (or hash structure, or B-tree on hash values). In this design, all participating single-table indexes share some leading key columns, but

typically not all key columns. Alternatively, one can extend some or all of the participating record types such that each record contains the union of all key columns. However, this alternative suffers from poor extensibility, because all existing records must be reformatted when a new single-table index and a new record type with a new key column are introduced, even if the new table contains no rows yet.

In the more flexible design, the sort key in a merged B-tree is more complex than in a traditional single-index B-tree. Each column value is prefixed by its domain (or domain identifier, or type, or type identifier), very similar to the well known notions of tagged union types. A new column type is the identifier of the single-table index to which a record belongs. Columns following the index identifier may omit their type identifiers. Figure 3 shows an example record for the *orders* table. While the design in Figure 3 lends itself to key normalization of the entire record, alternative designs tag each record with the identifier of its schema and compare records by alternating between two schemas and the two actual records.

Even the index of the *order details* table must include the leading field *customer number* in order to achieve the desired clustering of records in the B-tree. Such denormalization and field replication is required if two conditions hold. First, there must be multiple levels in the join graph that reassembles the complex objects from the index. Second, some intermediate tables must have unique keys of their own (a primary key that does not include a foreign key, or strong entities in the entity-relationship model) such as the table *orders*. In effect, the index for *order details* could be thought of as an index not on the table *order details* but as an index on the view joining *orders* and *order details*, and the maintenance of the index uses standard update plans for indexed (materialized) views.

Field value	**Field type**
"Customer number"	Domain identifier
Customer number	Actual value
"Order number"	Domain identifier
Order number	Actual value
"Table and index identifier"	A fixed domain identifier
Orders table, Customer Order index	References to catalog entries
Order date	Actual value
Order priority	Actual value
…	…

Figure 3. A sample record in a merged B-tree index.

Note also that the above index does not permit searching for *order* or *order details* rows by their *order number* only. Thus, it might be useful to introduce a separate index that maps *order numbers* to *customer numbers*. Moreover, either such an index is required for both the *orders* and the *order details* tables, or the query optimizer can exploit the index on *orders* even for *order details* searches on *order number* by analyzing primary and foreign key constraints, i.e., functional dependencies within and across tables.

Implementing master-detail clustering as multiple traditional single-table indexes merged into a single B-tree creates useful flexibility. For example, in a many-to-many relationship such as between suppliers and parts, it is possible to define two indexes on the table

representing the relationship, *part supply*, one on the foreign key *part number* and one on the foreign key *supplier number*, and then merge one of those indexes with the primary key index on the *parts* table and one with the primary key index on the *supplier* table. This design enables fast navigation from parts to suppliers and from suppliers to parts.

Even multiple indexes from a single table can be merged into a single B-tree, e.g., for a recursive relationship such as "reports to" among employees. In this example, the two indexes would be on *employee id* and *manager's employee id* – one of these could even be the clustered index, if desired.

In addition to I/O savings when assembling complex objects, merged indexes are also very useful for caching in query plans with nested iteration. In the earlier discussion of caches and replacement policies, we presumed two indexes, a control index that captures which combinations of parameter values have been processed before, and a data index that contains the results of nested query executions. Since both indexes use the correlation columns as the search key, it is obvious that they can be merged into a single B-tree, with equally obvious I/O savings.

4.4 Index intersection versus multi-dimensional indexes

For equality predicates on multiple columns, single-dimensional multi-column indexes are sufficient. For example, the predicate *where salary = 20 and bonus = 5* can be processed efficiently using a traditional multi-column B-tree on *employee (salary, bonus)*. If range predicates on multiple columns are frequent, e.g., *where salary between 20 and 30 and bonus between 3 and 9*, multi-dimensional index could be considered. Rather than implementing an entirely new index structure, a single-dimensional index combined with a space-filling curve can be very attractive, e.g., a B-tree for a Peano or Hilbert curve [B 97, RM 00, MJ 01].

There are effective techniques to reduce the I/O even in a single-dimensional multi-column index. For example, if there are only a few qualifying values of the first column (*salary* in the example index above), these values can be enumerated efficiently using the index, and then for each such value, index entries that qualify for the entire predicate can be searched for directly. This technique is exploited in some products by Tandem (now HP) and Sybase [LJB 95].

5 Data flow

The preceding discussion covered I/O, both avoiding it and speeding it up when it cannot be avoided. The present section covers data flow, i.e., how records should move in a complex query execution plan to achieve the desired effects on I/O. The subsequent section will cover control flow, i.e., how the desired data flow can be implemented with efficient and extensible mechanisms in single-threaded and multi-threaded query execution.

5.1 Batches of bindings

Sorting the outer input is only one possible method of many for improving the locality of data access in the inner plan. An alternative is to loosen the restriction that the inner execution plan can execute on behalf of only one outer row at a time. In other words, multi-

ple outer rows are bound and the inner plan executes only once for the entire set, hopefully improving execution efficiency. In general, it seems that batched execution is readily applicable if all intermediate results in the inner plan either are the same for all outer bindings in the batch or if distinct bindings in the outer bindings partition all stored tables and intermediate results in the inner plan into disjoint subsets of rows. In the first of those cases, the expected efficiencies are similar to caching; in the second case, they are similar to the advantages of set-oriented SQL queries compared to application-level row-at-a-time cursor operations. The size of the set may be determined simply by counting outer rows or by some column value in the outer input, very similar to value packets [K 80a].

5.1.1 Temporary storage for batches

In order to achieve these efficiencies, the batch of outer bindings must be accumulated and thus stored within the inner plan. Thus, it might be useful to introduce a new leaf in the inner plan (or multiple copies of the same leaf). This leaf is similar to a table stored in the database, except that it is a temporary table populated from above with a batch of outer bindings. If such a leaf is the direct input into a sort or (as build input) into a hash join, special implementations or modes of those operations may subsume the function of the new leaf. Thus, a sort operation may be the leaf of a query plan, with its input passed in as a batch of outer bindings; and a hash join may have only one input (the probe input).

For example, consider an inner plan that includes a filter operator with a predicate like "t.a = @a" where "t" is a table scanned in the inner plan and "@a" is a correlation bound by a nested iteration above the filter operator. For this filter to work for a batch of values for "@a", it might be advantageous to put all those values into a hash table. Thus, the filter becomes a hash join with the hash table built from outer correlation bindings. Note that this is very similar to a hash-based implementation of the filter operation for predicates of the form “t.a in (@a, @b, @c, @d, …, @z)” – some applications, in particular those with their own caches, employ very long “in” lists that benefit from a hash table in the filter operation, and a hash-based implementation of “in” filters is required to support those applications efficiently. Rather than implementing multiple forms of filter, it might be simpler to employ the hash join implementation for all filters that require memory for binding values; in the extreme case, with a single row in the build input.

Alternatively, implementation effort is reduced by implementing accumulation of outer bindings only in the new leaf operator and, where necessary, to use the leaf operator as the input for sort and hash join, e.g., as build input in the example. Using this simplification, both sort and hash join implementations are less complex and more maintainable, at probably only a moderate loss of execution efficiency.

One of the possible inefficiencies in an execution plan with batched bindings is that the outer rows must be visited twice within the outer plan, once to bind correlation values for the inner query and once to match the result of the inner query with the outer rows. Moreover, these rows might have to be stored twice, once in the outer execution plan and (at least) once in the inner execution plan. Fortunately, it is possible to share the actual store location or, alternatively, to bind entire outer rows such that the result of the inner query includes all columns required in the result of the nested iteration. The convenience and efficiency of storing and revisiting those outer rows might determine the optimal size of each batch of outer rows. One means to ease the implementation is to create a temporary

rary table, and to scan it repeatedly as required. If the batches are small, the temporary table will remain in the buffer pool and therefore never require I/O. If larger batches reduce more effectively the total cost of executing the inner plan and if those batches are large enough to spill from the buffer pool, one might as well compute and spool the entire outer input before initiating the inner input. This query execution plan achieves precisely the effect called *sideways information passing* or *magic decorrelation* in query optimization [SHP 96], which is closely related to semi-join reduction [BC 81].

5.1.2 Details of matching outer rows and inner results

If the inner query is invoked for batches of outer rows, each inner result row must be matched correctly with outer rows. If the outer correlation values may contain duplicates, either these duplicates must be removed prior to executing the inner query or they must be made unique by adding an artificial "rank" column. This notion of an artificial rank column is a standard technique when duplicate rows can create problems, e.g., when non-clustered indexes point to base rows in non-unique clustered indexes using search keys, when un-nesting and re-nesting non-first-normal-form relations [ÖW 92], when pausing and resuming a cursor or a series of next-page queries [L 01], etc.

The operation that matches outer rows and results from the inner query plan also needs to ensure correct scalar results. In SQL, a nested query may be used in place of a scalar value, e.g., "where T0.a = (select …)", as opposed to in place of a table expressions, e.g., "where T0.a in (select …)". In the latter case, any number of result rows from the inner query are permitted. In the former case, however, it is a run-time error if the inner query produces zero or multiple rows. These semantics must be preserved in any execution plan, including plans that employ batches of outer rows. Suitable algorithms are well known as "hybrid cache" [HN 96] or "flow distinct" [GBC 98].

5.1.3 Mixed batched and non-batched execution

If, in a complex inner plan, some operations permit batches and some do not, it is possible to invoke the entire inner plan with batches yet invoke some parts of the inner plan one outer row at a time. A simple technique to achieve this is adding to the inner plan a nested iteration (to bind one outer row at a time), with a leaf operator (to store an entire batch) as outer input and the row-at-a-time part of the inner plan as inner input.

Figure 4 illustrates this idea. It only shows the crucial components; a query optimizer would not choose this plan as it is as the inner plan is trivial. The upper nested iteration creates batches. Bindings are spooled in a temporary table in the leaf node. Once an entire batch has been bound, the upper nested iteration requests result rows from the lower one. To produce those, the lower iteration will obtain a single row from the spool operator, bind it to the inner plan, and pass result rows to the upper nested iteration.

5.2 Parallel execution

While batches can be very helpful for reducing I/O, they are practically required for efficient parallel execution of nested query plans, in particular nested query plans in distributed-memory parallel database servers. On the one hand, the cost of crossing boundaries between threads, processes, or machines is very high compared to invoking iterator

methods (a simple invocation of a virtual method). Thus, each crossing of boundaries should be leveraged for many rows. On the other hand, each thread executing the inner plan may be invoked by multiple threads executing the outer plan and controlling the nested iteration. Thus, inner result rows must be tagged with the outer row and the requesting outer thread. The required tags are very similar to the tags required in batched execution of inner plans; thus, any tagging required for executing inner plans for batches of outer rows introduces no additional overhead.

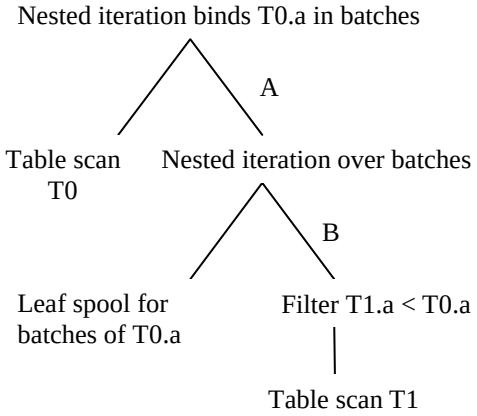

Figure 4. Using two nested iterations to create and dissect batches.

For these two reasons, inner query plans in parallel environments are best invoked for batches of outer rows. If the switch to batched execution cannot be accomplished automatically and transparently by the query execution engine, the query optimizer should attempt to rewrite the query using techniques described elsewhere as semi-join reduction and as magic decorrelation [BC 81, SHP 96]. If those techniques do not apply, batches of outer rows can be passed to the inner plan across thread, process, or machine boundaries, yet such batches are processed one row at a time using the technique shown in Figure 4 for complex inner plans in which some operations permit batch at a time operations and some do not.

6 Control flow

Given the large variety of algorithmic techniques described so far, implementing a state-of-the-art query execution engine might seem dauntingly complex. This section shows how to integrate the above techniques into an iterator interface very similar to the ones used in most commercial database systems [G 93]. The main operations for an iterator are, very similar to a file scan, *open*, *get next record*, and *close*. Each iterator invokes and therefore schedules all its input iterators or producers in demand-driven query execution. Alternatively, each iterator could invoke and schedule its output iterators or consumers in data-driven query execution, using an *accept next record* iterator method. Demand-driven query execution is preferable if multiple operators execute within a single process and if joins are more frequent than common subexpressions, i.e., when there are more operations with multiple inputs than operations with multiple outputs. On the other hand, data-

driven iterators seem preferable if iterators always communicate across thread boundaries or even machine boundaries and if common subexpressions are more frequent than joins.

6.1 Generalized spool iterator

In the following, we presume iterators based on demand-driven data flow. For common subexpressions, we presume a *spool* iterator that buffers its single input stream and provides the data to its multiple consumers at their desired pace. This operator exists in some form in most systems, for multiple purposes such as staging the result of a common subexpression or providing phase separation for protection from the Halloween problem [M 97]. As much as possible, in-memory buffers ought to be used, although temporary disk space must be available in case the input stream is very large and multiple consumers demand data at different rates. If the input data are sorted in a way that is useful later in the query plan, a sorted data structure such as a B-tree ought to be used.

The spool iterator can consume its input eagerly or lazily. The eager mode consumes its entire input immediately when *open*ed, similarly to a sort iterator. The lazy mode consumes its input only as fast as the fastest consumer requires spool output. The two modes can be a single implementation with a simple run-time switch.

This spool iterator can be generalized beyond these two prototypical modes for efficient nested iteration. For example, asynchronous I/O can be implemented using two fetch iterators, one providing prefetch hints to the store and the other actually obtaining the fetched records. These fetch iterators are separated by a spool iterator with a fixed-size record buffer to ensure that the store can always have multiple outstanding asynchronous I/O operations. This buffer can be run strictly first-in-first-out or it can use a priority queue (replacement selection) to pre-sort records. This mode of spool also can be useful in the outer input of a general nested iteration, as it may increase locality within the inner query without much cost. The buffer can be replenished from the input only when entirely empty (batch mode) or after each record passed to the consumer iterator (sliding window mode). A typical use case for the latter variant is enabling asynchronous I/O, while the former variant allows temporarily releasing resources in one part of a query plan for use in another part, e.g., releasing memory in an outer plan to make it available in an inner plan. However, managing resources explicitly and efficiently requires additional iterator methods to control resource usage, which will be discussed next.

6.2 New iterator methods

The principal mechanism to adapt the iterator interface to nested iteration is to extend the set of methods provided by all iterators. With almost half a dozen additional methods, a state diagram or a state transition table are the most concise means to convey all permissible method invocation sequences. A very helpful utility while developing a query execution engine is a “debug” iterator that can be inserted at any place in any query plan, where it enforces the state transition diagram, counts or prints intermediate result records, verifies sort order and other data properties predicted by the query optimizer, etc. If a version of this iterator is included in the final release of the query execution engine, it can compare anticipated and actual record counts or column distributions, and can thus also serve as the foundation for run-time feedback to the optimizer.

The required states capture whether an iterator is *open* or *closed*, whether or not all parameters are bound (*ready*), whether a result record has been produced but not yet released (*busy*), and whether or not the iterator is *paused*. Introducing the new iterator methods one by one will also clarify the purpose and semantics of these states.

Old state	Iterator method	New state	Comment
Closed	Open()	Open	
Open	Rewind()	Ready	Nothing unbound
Ready	GetNext()	Busy	Success
Ready	GetNext()	Ready	Failure, e.g., end
Busy	Release()	Ready	
Ready	Close()	Closed	
Ready	Rewind()	Ready	
Open	Close()	Closed	
Open	Bind()	Open	
Open	Unbind()	Open	
Ready	Unbind()	Open	
Ready	Pause()	Ready & paused	
Ready & paused	Resume()	Ready	
Busy	Pause()	Busy & paused	
Busy & paused	Resume()	Busy	
Ready & paused	Rewind()	Ready	
Ready & paused	Unbind()	Open	
Ready & paused	Close()	Closed	

Table 1. Iterator methods and state transitions.

In nested iteration, an additional iterator method *rewind* is required. For each outer row, the inner input is rewound, and the entire inner result can be recomputed. Most iterators implement this method by rewinding all their inputs; in this case, a rewind operation restarts an entire query plan. One exception to this rule is the *spool* iterator, which can simply fall back on its buffer to produce its output over and over. For simplicity, it might be tempting to require that the spool be eager; however, for efficiency, this is not a good design, because many nested queries terminate early, e.g., an “exists” or “top” subquery. Other “stop and go” operators might also implement efficient local rewind, e.g., sort and hash join. Unfortunately, hybrid hash join is more efficient than the original Grace hash join precisely because it does not write all its input to temporary files; thus, rewinding hybrid hashing might require more special cases than one is willing to implement, test, and support. In-memory hashing and completely external hashing (Grace hash join without dynamic destaging) lead more easily to efficient rewind operations. Sorting, on the other hand, can implement rewind operations quite readily; for example, only the final merge step needs to be restarted in an external merge sort.

For nested iteration with correlations, *bind* and *unbind* methods are required. If batched execution is not implemented, *bind* and *unbind* calls must be alternating strictly. In most cases, bind and unbind methods recursively traverse the entire inner plan, even across nested iteration operations within an inner plan. Implementations of the unbind method should not abandon all cached intermediate results, and the bind method should indicate

whether or not the new bindings invalidate intermediate results cached further up in the execution plan. In execution plans with complex inner plans, it is advantageous to annotate each sub-plan (or its root iterator) with the set of unbound parameters, and to avoid tree traversals where bind operations have no effect.

In order to implement batched execution, multiple *bind* operations without interleaved *unbind* operations must be permitted. In effect, a table of parameters is streaming down with the inner query plan, from the nested iteration towards the scan operations at the plan's leaves. The rows of this table, or the records in this data stream, may be buffered at one or multiple places in the inner plan, for later use. For example, an *exchange* operator might employ buffering, to be discussed shortly. As another example, a special spool iterator might be a plan leaf and buffer not an input stream but a table of parameters. During execution of the inner plan, the rows buffered in a spool iterator in *leaf* mode are like another input table for the inner plan – in fact, "magic" query execution and side-ways information passing presume that the entire set of parameters is a single large batch and serves as an additional input table in the inner query plan, which can readily be implemented using this *leaf* mode of spool.

In addition to the *close* method, which releases all resources but also terminates all iteration, a method is needed that lets a plan release resources for a while, in particular memory, yet later resume iteration. The methods to do so are *pause* and *resume*. For example, to pause a scan might release buffer pages used for double buffering, or a sort operation might write its input to disk even though the input was smaller than the memory available to the query. More interesting and more efficient, however, are iterators that do not immediate release their memory, but only register it as "waiting" with a query-wide memory manager. Only if the memory is indeed needed, and only as much as needed, will indeed be released. For example, if two sort operations feed a merge join, and their combined input is only slightly larger than the memory allocated to the query, only part of the first sort's input is written to a run file on disk, the remainder kept in memory as an in-memory run, and the two runs merged when sorted input is required by the merge join.

There are three typical use cases for the pause and resume methods. First and most importantly, a binary operation needs to open its inputs one at a time, and whichever input is opened first should pause while the second input is being opened. For example, a merge join fed by two sort operations can pause the first sort (which can therefore release its memory) while opening the second input. Earlier work presumed that between an iterator's *open* invocation and its first *get next* invocation would be the equivalent of a *pause*. Explicit iterator methods enable more control without requiring much more mechanism. The merge join in this example is typical for any bushy plan with resource contention among its branches.

Second, operations such as minor sort (sorting a pre-sorted stream on additional columns) or spool in batch mode consume their input and produce their output in batches. After a batch has been consumed, the input plan should release its resources if they can be used more productively elsewhere, e.g., in the inner plan while pausing the outer input. Thus, the spool iterator in batch mode should invoke the pause method on its input after filling its buffer with input rows.

Third, after a nested iteration iterator has bound a batch of outer rows to the inner query, the outer input may pause and release its resources for use other operations, in particular the inner plan. If both outer and inner plans are complex plans with memory-intensive bitmap, sort or hash operations, explicitly releasing and re-acquiring resources permits better performance with less memory contention.

Finally, as a performance feature, it may be useful to implement an iterator method that combines the bind-rewind-next-unbind sequence for a single record in a single method invocation, because there are many cases in actual query plans where it is known at compile-time that only a single record is needed. Typical examples include "exists" predicates and scalar nested queries, i.e., nested queries in place of scalar values. For the latter example, it is also very useful to implement a control flag in the iterator actually joining outer and inner rows that raises a run-time error if the inner result is empty or if it contains more than a single row (although the multiple-row case can be detected in a separate iterator, too, as discussed earlier).

6.3 Parallel execution

As discussed above, iterators can be data-driven or demand-driven. The former mechanism is most suitable for single-thread query execution, whereas the latter is often considered superior in parallel, multi-thread query execution. Fortunately, the two models can be combined, e.g., in the *exchange operator* that exposes and supports demand-driven iteration in its interaction with neighboring iterators within a thread but employs data-driven data flow (with flow control or "back pressure," when required) across thread and machine boundaries [G 96].

Parallel execution of nested queries presents two principal difficulties not encountered in single-threaded execution. First, crossing a thread, process, or even machine boundary is quite expensive, in a variety of ways. Most obviously, data packets must be assembled and, usually, the operating system and thread scheduler get involved. In addition, some threads must wait for data, and during the wait time, other threads will take over the CPU and more importantly the CPU cache. The second principal difficulty is that there might be multiple consumer threads that need to bind parameters for and invoke producer threads. In other words, each producer thread must produce data for every one of the consumer threads. The alternative design, assigning a pool of producer threads to each consumer thread, leads to an explosion in the number of threads, in particular if a query plan employs multiple levels of nesting – thus, this design does not scale to complex queries.

Differently than the iterator methods that bind and unbind parameters (or more precisely rows of parameter sets) as well as return results (or result rows), some iterator method affect the entire stream and the entire iterator state. For those, namely *rewind*, *pause*, and *resume*, the consumer side must wait for all producers to reach the appropriate state and invoke the same method. In other words, the producer side can, for example, only *rewind* after all consumers have requested it. Similarly, each producer must invoke all producers to rewind. For *pause* and *resume*, producer side must be active if at least one consumer requires it, and may pause only if all consumers require it. This coordination happens in each consumer – only the last *pause* and the first *resume* are passed from the consumer half of the exchange iterator to its input iterator.

7 Summary and conclusions

In summary, executing nested queries efficiently is not as simple as it might seem at first sight. Carefully implemented and managed, asynchronous I/O can improve execution performance by an order of magnitude in single-user operation and by a substantial margin in multi-user operation. Sorting correlation values in the outer query, entirely or only opportunistically using readily available memory, can improve locality and lookup performance in the inner query. Caching results of nested queries, including the fact that a result is empty for a given set of correlation values, can be an alternative or complementary technique, possibly contributing performance improvements of another order of magnitude. Since there may be many correlation values, data flow techniques such as processing in batches must apply not only to intermediate result data but also to streams of correlation values, particularly in parallel database systems. Finally, since inner queries can be very complex in their own right, outer query, inner query, and their result may compete for memory and other resources, which therefore must be mediated for maximal efficiency and query performance.

Crucial policy issues have not been resolved here, in particular allocation of memory and threads. Further issues include policy settings for asynchronous I/O, batching, and caching, e.g., LRU policies for cached results. There is no well-understood and agreed-upon solution for those issues that is simple enough to implement and test; general enough to cover nested iteration with multiple levels and multiple branches at each level, and with memory- and CPU-intensive sort-, hash-, and bitmap operations at multiple levels; and robust enough for database users to accept without the need or desire for manual tuning. Working through those issues towards a solution that combines generality with simplicity and robustness is a challenging research problem with immediate practical application. If this paper has contributed stimulation for this research, it has fulfilled its purpose.

Acknowledgements

Barb Peters' and César Galindo-Legaria's helpful suggestions were greatly appreciated.

References

[B 97] Rudolf Bayer: The Universal B-Tree for Multidimensional Indexing: General Concepts. Proc. Worldwide Computing and Its Applications. Lecture Notes in Computer Science 1274, Springer 1997: 198-209.

[BC 81] Philip A. Bernstein, Dah-Ming W. Chiu: Using Semi-Joins to Solve Relational Queries. JACM 28(1): 25-40 (1981).

[BMG 93] José A. Blakeley, William J. McKenna, Goetz Graefe: Experiences Building the Open OODB Query Optimizer. SIGMOD Conf. 1993: 287-296.

[CL 94] Richard L. Cole, Goetz Graefe: Optimization of Dynamic Query Evaluation Plans. SIGMOD Conf. 1994: 150-160.

[DNB 93] David J. DeWitt, Jeffrey F. Naughton, Joseph Burger: Nested Loops Revisited. PDIS 1993: 230-242.

[G 93] Goetz Graefe: Query Evaluation Techniques for Large Databases. ACM Computing Surveys 25(2): 73-170 (1993).

[G 96] Goetz Graefe: Iterators, Schedulers, and Distributed-memory Parallelism. Software - Practice and Experience 26(4): 427-452 (1996).

[GBC 98] Goetz Graefe, Ross Bunker, Shaun Cooper: Hash Joins and Hash Teams in Microsoft SQL Server. VLDB Conf. 1998: 86-97.

[GLS 93] Peter Gassner, Guy M. Lohman, K. Bernhard Schiefer, Yun Wang: Query Optimization in the IBM DB2 Family. IEEE Data Eng. Bull. 16(4): 4-18 (1993).

[GW 89] Goetz Graefe, Karen Ward: Dynamic Query Evaluation Plans. SIGMOD Conf. 1989: 358-366.

[H 78] Theo Härder: Implementing a Generalized Access Path Structure for a Relational Database System. ACM TODS 3(3): 285-298 (1978).

[HCL 97] Laura M. Haas, Michael J. Carey, Miron Livny, Amit Shukla: Seeking the Truth About ad hoc Join Costs. VLDB Journal 6(3): 241-256 (1997).

[HN 96] Joseph M. Hellerstein, Jeffrey F. Naughton: Query Execution Techniques for Caching Expensive Methods. SIGMOD Conf. 1996: 423-434.

[K 80] Won Kim: A New Way to Compute the Product and Join of Relations. SIGMOD Conf. 1980: 179-187.

[K 80a] Robert P. Kooi, The Optimization of Queries in Relational Databases. Ph.D. thesis, Case Western Reserve University, 1980.

[K 82] Robert Kooi, Derek Frankforth: Query Optimization in INGRES. IEEE Data Eng. Bull. 5(3): 2-5 (1982).

[KGM 91] Tom Keller, Goetz Graefe, David Maier: Efficient Assembly of Complex Objects. SIGMOD Conf. 1991: 148-157.

[L 01] Johan Larson, Computing SQL Queries One Webpage at a Time. Ph.D. thesis, University of Wisconsin – Madison, 2001.

[LJB 95] Harry Leslie, Rohit Jain, Dave Birdsall, Hedieh Yaghmai: Efficient Search of Multi-Dimensional B-Trees. VLDB Conf. 1995: 710-719.

[M 97] Paul McJones (ed.): The 1995 SQL Reunion: People, Projects, and Politics. Digital Systems Research Center, Technical Note 1997-018, Palo Alto, CA. Also http://www.mcjones.org/System_R.

[MHW 90] C. Mohan, Donald J. Haderle, Yun Wang, Josephine M. Cheng: Single Table Access Using Multiple Indexes: Optimization, Execution, and Concurrency Control Techniques. EDBT Conf. 1990: 29-43.

[MJ 01] Bongki Moon, H. V. Jagadish, Christos Faloutsos, Joel H. Saltz: Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. IEEE TKDE 13(1): 124-141 (2001).

[OOW 93] Elizabeth J. O'Neil, Patrick E. O'Neil, Gerhard Weikum: The LRU-K Page Replacement Algorithm For Database Disk Buffering. SIGMOD Conf. 1993: 297-306.

[ÖW 92] Z. Meral Özsoyoglu, Jian Wang: A Keying Method for a Nested Relational Database Management System. ICDE 1992: 438-446.

[RM 00] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, Rudolf Bayer: Integrating the UB-Tree into a Database System Kernel. VLDB Conf. 2000: 263-272.

[S 81] Michael Stonebraker: Operating System Support for Database Management. CACM 24(7): 412-418 (1981).

[SHP 96] Praveen Seshadri, Joseph M. Hellerstein, Hamid Pirahesh, T. Y. Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J. Stuckey, S. Sudarshan: Cost-Based Optimization for Magic: Algebra and Implementation. SIGMOD Conf. 1996: 435-446.