Scalable Computation of Acyclic Joins

(Extended Abstract)

Anna Pagh
annao@itu.dk

Rasmus Pagh
pagh@itu.dk

IT University of Copenhagen
Rued Langgaards Vej 7, 2300 København S
Denmark

ABSTRACT

The *join* operation of relational algebra is a cornerstone of relational database systems. Computing the join of several relations is NP-hard in general, whereas special (and typical) cases are tractable. This paper considers joins having an *acyclic join graph*, for which current methods initially apply a *full reducer* to efficiently eliminate tuples that will not contribute to the result of the join. From a worst-case perspective, previous algorithms for computing an acyclic join of k fully reduced relations, occupying a total of $n \geq k$ blocks on disk, use $\Omega((n+z)k)$ I/Os, where z is the size of the join result in blocks.

In this paper we show how to compute the join in a time bound that is within a constant factor of the cost of running a full reducer plus sorting the output. For a broad class of acyclic join graphs this is $O(\mathrm{sort}(n+z))$ I/Os, removing the dependence on k from previous bounds. Traditional methods decompose the join into a number of binary joins, which are then carried out one by one. Departing from this approach, our technique is based on computing the size of certain subsets of the result, and using these sizes to compute the location(s) of each data item in the result.

Finally, as an initial study of cyclic joins in the I/O model, we show how to compute a join whose join graph is a 3-cycle, in $O(n^2/m + \mathrm{sort}(n+z))$ I/Os, where m is the number of blocks in internal memory.

Categories and Subject Descriptors

H.2 [**Database management**]: Systems—*Relational databases*

General Terms

Algorithms, Performance, Reliability, Theory

Keywords

Relational algebra, acyclic join, algorithm, external memory

PODS'06, June 26–28, 2006, Chicago, Illinois, USA.

1. INTRODUCTION

The *relational model* and *relational algebra*, due to Edgar F. Codd [3] underlies the majority of today's database management systems. Essential to the ability to express queries in relational algebra is the *natural join* operation, and its variants. In a typical relational algebra expression there will be a number of joins. Determining how to compute these joins in a database management system is at the heart of *query optimization*, a long-standing and active field of development in academic and industrial database research. A very challenging case, and the topic of most database research, is when data is so large that it needs to reside on secondary memory. In that case the performance bottleneck is the number of block transfers between internal and external memory needed to perform the computation. We will consider this scenario, as formalized in the I/O model of Aggarwal and Vitter [1]. We remark that increasingly, the caching system on the RAM of modern computers means that algorithms developed for external memory, with their parameters suitably set, have a performance advantage on internal memory as well.

In contrast to much of the database literature, our emphasis will be on *worst case* complexity. While today's database management systems usually perform much better than their worst case analysis, we believe that expanding the set of queries for which good worst case performance can be guaranteed would have a very positive impact on the reliability, and thus utility, of database management systems.

The worst case complexity of computing a binary join in the I/O model is well understood in general – essentially the problem is equivalent to sorting [1]. When more than two relations have to be joined, today's methods use a sequence of binary join operations. Since the natural join operator is associative and commutative, this gives a great deal of choice: We may join k relations in $2^{\Omega(k)}k!$ different ways (corresponding to the unordered, rooted, binary trees with k labeled leaves). The time to compute the join can vary a lot depending on which choice is made, because some join orders may give much larger intermediate results than others. Research in query optimization has shown that finding the most efficient join order is a computationally hard problem in general (see e.g. [6, 8]). The best algorithms use sophisticated methods for estimating sizes of intermediate results (see e.g. [4]).

More theoretical research has considered the question of what joins are tractable. The problem is known to be NP-

hard in general [10], but the special case of joins having an *acyclic join graph* has been known for many years to be solvable in time polynomial in the sizes n and z of the input and output relations [15, 16]. In internal memory, Willard [14] has shown how to reduce the complexity of acyclic joins involving n words of data to $O(n(\log n)^{O(1)} + z)$ if the number k of relations is a constant, even when the join condition involves arbitrary equalities and inequalities. However, Willard's algorithm does not have a good dependence on k – the time is $\Omega(kn)$ for natural join, and for general expressions the dependence on k is exponential.

The main *conceptual* contribution of this paper is that we depart from the idea of basing join computation on a sequence of binary joins. This is an essential change: We will show that any method based on binary joins, even if it always finds the optimal join order, has a multiplicative overhead of $\Omega(k)$ compared to the time for reading and writing all input and output data.

Our main *technical* contribution is a worst case efficient algorithm for computing acyclic joins, whose complexity is independent of the number of relations. Instead, the complexity can be bounded in terms of the amount of data that goes into all possible binary joins among the relations, counting only data in shared attributes, plus the size of the input and output. For example, if each attribute occurs $O(1)$ times, the complexity is within a constant factor of the cost of sorting the input and output relations. The algorithm is simple enough to be of practical interest.

1.1 Background and previous work

1.1.1 Query optimizer architecture.

Query processing in relational databases is a complex subject due to the power of query languages. Current query optimizers use an approach first described in [12], based on relational algebra [3]. The basic idea is to consider many equivalent relational algebra expressions for the query, and base the computation on the expression that is likely to yield the smallest execution time. The best candidate is determined using heuristics or estimates of sizes of intermediate results. When computing a join, the possible relational algebra expressions correspond to expression trees with k leaves/relations. Some query optimizers consider only "left-deep join trees", which are highly unbalanced trees, to reduce the search space. In some cases such an approach has led to a large overhead, intuitively because data from the first joins had to be copied in all subsequent joins. Approaches based on more balanced "bushy trees" have yielded better results in such cases. (See [9] for a discussion on left-deep versus bushy trees.) Note, however, that even in a completely balanced tree with k leaves, most leaves will be at depth $\log k - O(1)$. Thus, data from the corresponding relations needs to be read (and written) in connection with about $\log k$ joins before it can appear in the result. Furthermore, there exist joins for which the best tree is highly unbalanced (see Section 3.1).

1.1.2 Speedup techniques.

Pipelining is sometimes used to achieve a speedup by using any available memory to start subsequent joins "in parallel" with an on-going join, thus avoiding to write an intermediate result to disk. However, the speedup factor is at most constant unless an involved relation (or intermediate result) fits in internal memory. Binary join algorithms based on *hashing* can give better performance in the case where one relation is much smaller than the other, but again this is at most a constant factor speedup.

Indexing, i.e., clever preprocessing of individual relations, is another way of speeding up some joins. The gain comes from bypassing a sorting step, and in many practical cases by being able to skip reading large parts of a relation. However, from a worst-case perspective the savings on this technique is not more than the time for sorting the individual relations. Thus the previously mentioned $\Omega(k)$ factor persists. Special *join indexes* that maintain information on the join of two or more relations are sometimes used in situations where the same relations are joined often. However, maintaining join indexes for many different sets of relations is not feasible (because of time and space usage), so this technique is impractical for solving the general, *ad-hoc* join problem.

1.1.3 Acyclic joins.

Some joins suffer from extremely large intermediate results, no matter what join ordering is chosen. This is true even for joins that have no tuples in the final result. In typical cases, where the *join graph* is *acyclic* (see Section 2.1 for definitions), it is known how to avoid intermediate results that are larger than the final result by making $k - 1$ semijoins with the purpose of eliminating "dangling tuples" that can never be part of the final result. Such an algorithm, eliminating all tuples that are not part of the final result, is known as a "full reducer". Using a full reducer based on semijoins is, to our knowledge, the only known way of avoiding intermediate results much larger than the final result. Thus, using a full reducer is necessary in current algorithms if good worst-case performance is required. As our main algorithm mimics the structure of a full reducer, we further describe its implementation in Section 4.

1.1.4 Further reading.

For a recent overview of query optimization techniques we refer to [7]. We end our discussion of query optimization with a quote from the newest edition of a standard reference book on database systems [5]: *"Relational query optimization is a difficult problem, and the theoretical results in the space can be especially discouraging. [...] Fortunately, query optimization is an arena where negative theoretical results at the extremes do not spell disaster in most practical cases. [...] The limitation of the number of joins is a fact that users have learned to live with, [...]".*

1.2 Our main result

In this paper we present a new approach to computation of multiple joins for the case of acyclic join graphs (see Section 2.1 for definitions). The new technique gives a worst case efficient, deterministic algorithm, that is free from the dependence on the number of relations exhibited by previous techniques.

THEOREM 1. *Let $k \leq m$ relations, having an acyclic join graph, be given. We can compute the natural join of the relations, occupying z blocks on disk, in $O(t_{\text{reduce}} + \text{sort}(z))$ I/Os, where t_{reduce} is the time for running a semijoin based full reducer algorithm on the relations, and $\text{sort}(z)$ is the time for sorting z blocks of data.*

Our use of the sort() notation is slightly unusual, in that we have not specified how many data items can be in a block,

and the complexity of sorting depends on the total number of items, not just on the number of blocks. However, for reasonable parameters (e.g., if the number of items in a block is bounded by $m^{1-\Omega(1)}$), the actual number of items does not affect the asymptotic complexity. Note that the statement above allows for variable-length data in the relations.

The value of t_{reduce} depends on the join graph, and on the data in the relations. Essentially, a full reducer performs a number of binary semijoins on (subsets of) the relations, each of which boils down to sorting the data in the common attributes of the two relations. The complexity can thus be bounded in terms of the amount of data in common attributes of these semijoins. It is possible to construct examples where $t_{\text{reduce}} = \Omega(k \operatorname{sort}(n))$. However, in most "reasonable" cases the value is $O(\operatorname{sort}(n))$. For example, if each attribute occurs only in a constant number of relations (independent of k) we have $t_{\text{reduce}} = O(\operatorname{sort}(n))$.

In *internal memory* all sorting steps of our algorithm can be replaced by hashing (since it suffices to identify identical values). Hence, in those cases where $t_{\text{reduce}} = O(\operatorname{sort}(n))$, we get a randomized, expected linear time algorithm for internal memory.

1.3 Overview of paper

Section 2 contains definitions and notation used throughout the paper, as well as a description of our model and assumptions. Section 3 discusses *star schemas*, a case in which algorithms based on binary joins perform poorly in the worst case. We then present a worst-case efficient algorithm for star schemas, to illustrate the main ideas of our general algorithm. We survey known ways of implementing a full reducer in Section 4, to be able to describe our algorithm relative to them. In Section 5 we present our algorithm for joins with acyclic join graphs. Finally, in Section 6 we present our algorithm for joins whose join graph is a 3-cycle.

2. PRELIMINARIES

2.1 Definitions and notation

Let A be a set, called the set of *attributes*, and let for each attribute $a \in A$ correspond a set $\mathtt{dom}(a)$ called the *domain* of that attribute. In the context of databases, a *relation* R with attribute set $A_R \subseteq A$ is a set of functions from A_R, such that all function values on $a \in A_R$ belong to $\mathtt{dom}(a)$. We deal only with relations having a *finite* set of attributes and tuples. Since a function can be represented as a tuple of $|A_R|$ values, the functions are referred to as *tuples* (this is also the usual definition of a relation).

Consider two relations R_1 and R_2, having attribute sets A_{R_1} and A_{R_2}, respectively. The *natural join* of R_1 and R_2, denoted $R_1 \bowtie R_2$, is the relation that has attribute set $A_{R_1} \cup A_{R_2}$, and contains all tuples t for which $t|_{A_{R_1}} \in R_1$ and $t|_{A_{R_2}} \in R_2$. In other words, $R_1 \bowtie R_2$ contains all tuples that agree with some tuple in R_1 as well as some tuple in R_2 on their common domain. The restriction of the tuples in a relation R to attributes in a set A' (referred to as *projection*) is denoted $\pi_{A'}(R)$ in relational algebra. Thus, $R_1 \bowtie R_2$ is the maximal set R of tuples for which $\pi_{A_{R_1}}(R) = R_1$ and $\pi_{A_{R_2}}(R) = R_2$. A more general kind of join, called an *equijoin*, allows tuples to be combined based on an arbitrary set of equality conditions. However, it is easy to see that any equi-join can be reduced to a natural join by "renaming attributes". The *semijoin* $R_1 \ltimes R_2$ is a shorthand for $\pi_{A_{R_1}}(R_1 \bowtie R_2)$. In words, it computes the tuples of R_1 that contribute to the join $R_1 \bowtie R_2$.

It is easy to see that natural join is associative and commutative. Thus is makes sense to speak of the natural join of relations $R_1, \ldots, R_k$, denoted $R_1 \bowtie \cdots \bowtie R_k$. We will consider the *join graph*, which is the hypergraph having the set of all attributes as vertices, and the sets $A_{R_1}, \ldots, A_{R_k}$ as edges. The join graph is called *acyclic* if there exists a tree with k vertices, identified with the relations $R_1, \ldots, R_k$ (and thus associated with the sets $A_{R_1}, \ldots, A_{R_k}$), such that for any i and j, $A_{R_i} \cap A_{R_j}$ is a subset of all attribute sets on the path from R_i to R_j. It is known how to efficiently determine whether a join graph is acyclic, and construct a corresponding tree structure if it is [16].

We briefly discuss the notion of acyclicity. For a join graph to be *cyclic* a necessary (but not sufficient) condition is that there exists a *cycle*, i.e., a sequence of attribute sets $A_{R_{i_0}}, A_{R_{i_1}}, \ldots, A_{R_{i_{\ell-1}}}$, $\ell \geq 3$ such that:

- $A_{R_{i_j}} \cap A_{R_{i_{(j+1) \bmod \ell}}} \neq \emptyset$, for $j = 0, \ldots, \ell - 1$
- No set in the sequence is contained in another.

In the case where all attribute sets have size two (a normal graph), the join graph is acyclic exactly if it is a forest (collection of trees).

2.2 Model and assumptions

Our algorithms are for the I/O model of computation [1], the classical model for analyzing disk-based algorithms. The complexity measure of the model is the number of block transfers between internal and external memory. We let n and m denote, respectively, the number of input blocks (containing the relations to be joined) and the number of disk blocks fitting in internal memory. Since we deal with relations that can have attribute values of varying sizes, we do not have fixed-size data items, as in many algorithms in this model. Therefore the parameter B, usually used to specify the "number of items in a disk block" does not apply. We will simply express our bounds in terms of the number of disk blocks that are inputs to some sorting step, and the number of disk blocks read and written in addition to sorting. To simplify matters, we make the reasonable assumption that $m \geq k$. In particular, we can assume without loss of generality that $n > k$.

As for the input and output representation, we assume that all relations are stored as a sequence of tuples. A tuple, in turn, is a sequence of attribute values (in some standard order). The encoding of values of an attribute a is given by an arbitrary, efficiently decodable prefix code for the domain $\mathtt{dom}(a)$.

3. PRIMER: STAR SCHEMAS

As a warm-up for our general algorithm, we consider the special case of star schemas. We first show that algorithms based on binary joins perform poorly in the worst case, and then show a more efficient algorithm.

3.1 Worst case behavior of current algorithms

In this section we give, for any k and z, where $k < z^{1-\Omega(1)}$ and $m = o(z)$, an example of an acyclic join $R_1 \bowtie \cdots \bowtie R_k$ where:

- The result of the join occupies z blocks on disk.
- The relations occupy $n \leq z$ blocks on disk.
- Any algorithm computing the join using a sequence of binary joins will use $\Omega(zk)$ I/Os.

We consider a *star schema* where the sets $A_{R_2}, \ldots, A_{R_k}$ of attributes are of size 2 and disjoint, but each intersecting A_{R_1} in one attribute. All domains have encodings of the same, fixed length (large enough to encode unique values for all tuples). Our example will be such that each tuple in R_1 matches exactly one tuple in each of $R_2, \ldots, R_k$. In particular, the size z of the result of the join is twice the size of R_1 (in disk blocks). We let each of the relations $R_2, \ldots, R_k$ occupy at most $z/(2k)$ blocks each. In particular, each of $R_2, \ldots, R_k$ consist of $z^{\Omega(1)}$ blocks. Note that it is possible to choose the contents of the relations such that the join is fully reduced, assuming that k is larger than some constant (which we may assume without loss of generality).

We will argue that no matter how the join is computed, the total size of the intermediate results is $\Omega(zk)$ blocks. This means that most intermediate results cannot fit in internal memory, and in particular $\Omega(zk)$ I/Os are needed. Consider an arbitrary expression tree for the join. If there is a subexpression with a result of z^2 blocks or more, we are done. Otherwise, we argue that the path from the leaf containing R_1 to the root has length $\Omega(k)$. This is because any subexpression that does *not* include R_1 must involve $O(1)$ relations, since these subexpressions are cartesian products of relations having at least $z^{\Omega(1)}$ tuples (using that any block is large enough to hold at least one tuple). Thus, there are $\Omega(k)$ intermediate results of size $\Omega(z)$, finishing the argument.

Note that since $n \leq z$, we can alternatively state the lower bound as $\Omega((n+z)k)$ I/Os.

3.2 Worst-case efficient algorithm

We now describe a simple, worst-case efficient algorithm for star schemas that introduces some of the main ideas of our general algorithm. The characteristic feature of a star schema is that any two attribute sets are disjoint, except for pairs involving R_1, the "center" of the star, which shares one or more attributes with any other relation.

3.2.1 Simple case

Let us first consider the case in which the attributes of R_1 are foreign key references to $R_2, \ldots, R_k$. That is, every tuple of R_1 matches exactly one tuple in each of $R_2, \ldots, R_k$. Note that this holds for the worst-case example described above.

First of all, we know that there will be $|R_1|$ tuples in the result. We will output these tuples in the same order as the corresponding tuples appear in R_1. To this end, we consider the relation R_1' which is R_1 with an extra attribute `row` added, containing the "row number" of each tuple. Now, for each $i \geq 2$, compute

$$T_i = \pi_{(A_{R_1} \cap A_{R_i}) \cup \{\texttt{row}\}}(R_1') \ .$$

This can be done simultaneously for all i in a single scan of R_1, because of the assumption $m \geq k$. Notice that the number of I/Os used is $O(n)$. Then, for $i \geq 2$ compute

$$U_i = \pi_{(A_{R_i} \setminus A_{R_1}) \cup \{\texttt{row}\}}(T_i \bowtie R_i) \ .$$

This involves binary joins on data of total size $O(n)$.

It is not hard to see that the result of the star join can now be computed using the following relational algebra expression:

$$\pi_{\cup_i A_{R_i}}(R_1' \bowtie U_2 \bowtie \cdots \bowtie U_k) \ .$$

The relations in this join have exactly one common attribute, `row`. This means that the join can be computed by sorting all the participating relations according to `row`, and forming output tuples from the sets of k adjacent tuples with the same `row` value.

Conceptually, what happened was that we labeled (in U_i) all tuples of $R_2, \ldots, R_k$ with the location in the result in which they should appear. This made it possible to assemble the final result with no intermediate results.

3.2.2 General case

Consider now the general case in which a tuple in R_1 can match any number of tuples (including zero) in each of $R_2, \ldots, R_k$. We introduce R_i' for $i \geq 2$, which is R_i with an extra attribute `row2` containing the row number of each tuple. Compute the relations T_i as in the simple case, and let, for $i \geq 2$

$$U_i' = \pi_{\{\texttt{row},\texttt{row2}\}}(T_i \bowtie R_i') \ .$$

Now there can be any number of tuples in U_i' with a given value in the `row` attribute. By sorting U_i' for all i according to `row` we can determine for each tuple in R_1 the number of matching tuples in each of the other relations. This allows us to compute the number s_t of *occurrences* of a given tuple $t \in R_1$ in the final result, i.e., the number of tuples that would no longer be in the join result if t was removed from R_1. This is the product of the number of matches in each of the other relations. We will produce the output such that all tuples containing row number 1 from R_1 come first, followed by tuples containing row number 2, etc. By computing a prefix sum of the s_t values, we can determine for each $t \in R_1$ in which output tuples (i.e., in which row numbers) it should occur.

Finally this information needs to be propagated to the other relations. For a given tuple $t \in R_1$ with $s_t > 0$ occurrences we decide in an arbitrary way how the corresponding s_t tuples (essentially a cartesian product of the matching tuples from $R_2, \ldots, R_k$) should be formed. This gives for each $(j_1, j_2) \in U_i'$ a list of result tuples in which row number j_1 from R_1 and row number j_2 from R_i should occur. Sorting these according to `row2` we get for each tuple $t \in R_i$, $i \geq 2$, the set L_t of output tuples in which it should occur. Creating $|L_t|$ duplicates of t, marked with the row numbers in L_t, and sorting these by row number gives the desired result.

As in the simple case, the amount of data processed in scans and joins is $O(n)$ blocks, except for the final sorting step which involves $O(z)$ blocks of data. In conclusion, the algorithm uses $O(\text{sort}(n+z))$ I/Os.

4. FULL REDUCERS

We consider the join $R_1 \bowtie \cdots \bowtie R_k$, where the join graph is acyclic. The goal of a full reducer is to remove all tuples from the relations that do not contribute to the result of the join, i.e., whose removal would leave the join result the same. Here, we will describe a well-known construction, namely how to implement a full reducer using a sequence of semijoins. (This seems to be the only known approach.)

The description of our join algorithm in Section 5 will be based on the description below, as our algorithm has the same overall structure. We leave open several choices that affect the efficiency (but do not affect the correctness) of the full reducer, such as determining the best order of computing the semijoins. This is an interesting research question of its own (see e.g. [11]), but the only goal here is to *relate* the complexity of our join algorithm to that of a semijoin based full reducer.

4.1 First phase

The computation is guided by a tree computed from the join graph, satisfying the requirement stated in Section 2.1. We make the tree rooted by declaring an arbitrary node to be the root. In the *first* phase of the algorithm, the relations are processed according to a *post-order* traversal of the tree. We renumber the relations such that they are processed in the order $R_k, R_{k-1}, R_{k-2}, \ldots$, and assign the tuples of each relation R_i numbers $1, 2, \ldots, |R_i|$ according to their position in the representation of R_i. When processing a non-root relation R_j, a semijoin is performed with its parent relation R_i to eliminate all tuples of R_i that do not agree with any tuple of R_j on their common attributes. This can be done by sorting R_i and $\pi_{A_{R_i} \cap A_{R_j}}(R_j)$ according to the values of attributes in $A_{R_i} \cap A_{R_j}$, and merging the two results. The semijoin produces a subset of R_i that will replace R_i in the subsequent computation (i.e., we remove dangling tuples straight away, but keep referring to the relation as R_i).

4.2 Second phase

The *second* phase of the full reducer algorithm processes the relations according to a *pre-order* traversal (e.g. in the order $R_1, R_2, R_3, \ldots$). It is analogous to the first phase, but with the roles of parent and child swapped. When processing a non-leaf relation R_i, we replace each of its children R_{j_ℓ}, $\ell = 1, \ldots, r$, by the result of the semijoin $R_{j_\ell} \ltimes R_i$. This can be done efficiently by first computing $T_\ell = \pi_{A_{R_i} \cap A_{R_{j_\ell}}}(R_i)$ (for $\ell = 1, \ldots, r$ at the same time), and then computing the desired result as $R_{j_\ell} \ltimes T_\ell$. It is easy to see that the complexity of the second phase is no larger than that of the first phase.

We refer to the exposition in [13] for an argument that performing the semijoins as described does indeed fully reduce the relations.

4.3 Alternative first phase

In the first phase it will in some cases be more efficient to consider the semijoins involving all children of R_i, call them $R_{j_1}, \ldots, R_{j_r}$, at the same time, as follows:

1. For $\ell = 1, \ldots, r$ compute the extended projection T'_ℓ which is equal to $\pi_{A_{R_i} \cap A_{R_{j_\ell}}}(R_i)$ except that each tuple is extended with an extra attribute `row`, whose value is the position of the corresponding tuple in the representation of R_i (i.e., the number of tuples is $|R_i|$). This can be done for all ℓ in a single scan of R_i.

2. For $\ell = 1, \ldots, r$ compute the set of row numbers in R_i matching at least one tuple in R_{j_ℓ}, i.e., $\pi_{\texttt{row}}(T'_\ell \ltimes \pi_{A_{R_i} \cap A_{R_{j_\ell}}}(R_j))$. The complexity of this is dominated by the semijoin, which is computed by sorting the two involved relations.

3. Sort the sets from step 2, compute their intersection, and finally scan through R_i to eliminate all rows whose number is not in the intersection.

We will not discuss in detail when to choose the latter alternative (let alone a mix of the two alternatives), but highlight some cases in which it is particularly efficient. Consider the amount of data in the common attributes of R_i and each of its children, counting an attribute of R_i x times if it is an attribute of x children. If this quantity is bounded by the size of the representation of R_i and its children (times some constant), then the complexity of this step of the algorithm is bounded by a constant times the complexity of sorting the involved relations. If this is true for all i, the entire first phase is performed in sorting complexity.

In the following, we let t_{reduce} denote the currently best known I/O complexity, for the join in question, of a full reducer algorithm of the form described above.

5. ACYCLIC JOIN ALGORITHM

We consider computation of $R_1 \bowtie \cdots \bowtie R_k$, where the join graph is acyclic. To simplify the exposition, we consider the situation in which no attributes are from "small" domains. Specifically, we assume that any domain is large enough to encode a pointer to a tuple in the input and output. We will need this assumption in the complexity analysis since it means that the size of a pointer/counter is bounded by the size of an attribute value. In Section 5.6 we sketch how to modify the algorithm to handle also small domains within the same I/O bound.

5.1 Overview

Our algorithm first applies a full reducer to the relations, such that all remaining tuples will be part of the join result. Then, if there exists a relation R_i whose set of attributes is a subset of the attributes of another relation in the join, it will not affect the join of the fully reduced relations, and we may eliminate it. Thus, without loss of generality, we assume that no such relation exists.

We renumber the relations as described in Section 4, and number tuples according to their position in the representation of relations, using the induced order as an ordering relation on the tuples. Any tuple in the result will be a combination of tuples $t_1, \ldots, t_k$ from $R_1, \ldots, R_k$, respectively. The order of the result tuples will be *lexicographic*, in the following sense: For two result tuples t and t', being combinations of $t_1, \ldots, t_k$ and $t'_1, \ldots, t'_k$, respectively, $t < t'$ iff there exists i such that $t_1 = t'_1, \ldots, t_{i-1} = t'_{i-1}$ and $t_i < t'_i$.[1]

After applying the full reducer, our algorithm goes on to a *counting phase* (Section 5.2) that works bottom-up in the tree, mirroring the first phase of the full reducer algorithm. At each node R_i this phase considers the join corresponding to the *subtree* rooted at R_i, and computes for each tuple $t \in R_i$ the number of tuples of the subtree join, of which t is part. (For comparison, the full reducer algorithm just keeps track of whether these numbers are zero or nonzero.) The counts are then used in an *enumeration phase* (Section 5.3) that works top-down, but is otherwise rather different from the second phase of the full reducer algorithm. This phase computes for each tuple $t \in R_i$ the positions of tuples in

[1]Note the underlying assumption that relations are sets, i.e., have no duplicate tuples. However, our algorithm will work in the expected way in presence of duplicate tuples.

the result relation of which this tuple is part, i.e., whose projection onto A_{R_i} equals t. The positions will correspond to the lexicographic order of the result tuples. Thus, the output from the enumeration phase is a set of pairs (p, t), where the tuple t should be part of row number p in the result of the join. Sorting these pairs according to p, the k tuples having a particular row number will end up together, and it is trivial to form the joined tuples in a single scan. Below we give the details of the counting and enumeration phases. Figure 5.3 shows an example of how the algorithm runs on a particular input.

5.2 Counting phase

Consider a relation R_i that has relations $R_{j_1}, \ldots, R_{j_l}$ as descendants in the tree. For every tuple $t \in R_i$ we wish to compute the size s_t of $\{t\} \bowtie R_{j_1} \bowtie \cdots \bowtie R_{j_l}$. It might be useful at this point to recall the first part of the algorithm in Section 3.2.2 which does the same kind of counting in a simplified setting. The computation is done for $i = k, k-1, k-2, \ldots$, i.e., going bottom-up in the tree. At the leaves, all tuples have a count of 1. In the general step, assume $R_{j_1}, \ldots, R_{j_r}$ are the children of R_i, and that the tuple counts for these relations have been computed. The counts for tuples in R_i can now be computed by a slight extension of the procedure for computing semijoins involving R_i and its children in the full reducer. The changes are:

- Extend tuples in $T_\ell = \pi_{A_{R_i} \cap A_{R_{j_\ell}}}(R_{j_\ell})$, $\ell = 1, \ldots, r$, with an extra attribute `s` that contains, for each tuple $t \in T_\ell$, the sum of counts for all tuples in R_{j_ℓ} that agree with T_ℓ on their common attributes, i.e., whose projection onto $A_{R_i} \cap A_{R_{j_\ell}}$ equals t. This is easily done by sorting T_ℓ. In words, this gives the count of tuples in the join of the relations in the subtree rooted by R_{j_ℓ} for every value on the common attributes $A_{R_i} \cap A_{R_{j_\ell}}$.
- The count s_t for a tuple $t \in R_i$ is the product of the `s`-values of the matching tuples in the relations $\pi_{A_{R_i} \cap A_{R_{j_\ell}}}(R_{j_\ell})$, for $\ell = 1, \ldots, r$. The `s`-values are easily retrieved by performing the same sorting steps as needed for the semijoins, no matter which of the two variants described in Section 4 is used. Also, it is an easy matter to keep track of the original positions of tuples of R_i, such that the list of counts can be ordered accordingly.

Figure 5.3 shows the result of the counting phase on the example relations.

5.3 Enumeration phase

In this phase we annotate each tuple t in the input relations with the row numbers of the final result of which it should be part, generalizing the second part of the algorithm in Section 3.2.2. More specifically, we compute for each relation R_i a sorted list of disjoint intervals of result row numbers, where each interval has some tuple $t \in R_i$ associated (the actual tuple, not just its position in R_i). The computation proceeds top-down, considering the relations in the order $R_1, R_2, R_3, \ldots$. From the counting phase we know the number of occurrences s_t of each tuple $t \in R_1$ (the root relation) in the final result. The positions of the tuples in the lexicographic order of the result relation are intervals whose boundaries are the prefix sums of the s_t values. (See example in Figure 5.3.)

In the general step, when considering a non-leaf relation R_i we have a number of intervals, each associated with a tuple $t \in R_i$, and we wish to compute the intervals for the children of R_i. The invariant is that any interval associated with t has length s_t. Again, let $R_{j_1}, \ldots, R_{j_r}$ denote the children of R_i, $j_1 < \cdots < j_r$. The first thing is to retrieve for each tuple t, and $\ell = 1, \ldots, r$, the row numbers (in the representation of R_{j_ℓ}) of the matching tuples $R_{j_\ell} \ltimes \{t\}$, along with their counts. This can be done by first sorting according to common attributes, merging, and then performing another sorting to get the information in the same order as the intervals. The result tuples in the interval of t, restricted to the attributes $A_{R_{j_1}} \cup \cdots \cup A_{R_{j_r}}$, is the multiset cartesian product of the sets $R_{j_\ell} \ltimes \{t\}$, $\ell = 1, \ldots, r$, where the multiplicity of a tuple is its count, ordered lexicographically. This means that we can form the correct intervals for each relation, associated with row numbers.

5.4 Correctness

We now sketch the argument for correctness of our algorithm. It builds on the following observation on acyclic joins: If, for some relation R_i, we remove the vertices A_{R_i} from the join graph, this splits the graph into several connected components (some components may be the empty set). For $t \in R_i$, the set of result tuples containing t (i.e., $R_1 \bowtie \cdots \bowtie R_{i-1} \bowtie \{t\} \bowtie R_{i+1} \bowtie \cdots \bowtie R_k$) is the cartesian product of the tuples matching t in the join of the relations in each of these components. Consider the tree derived from the (original) join graph. If we split this into subtrees by removing the node corresponding to R_i, each part corresponds to one or more connected components in the join graph with A_{R_i} removed (this follows from the definition of acyclicity). Thus, the result tuples containing $t \in R_i$ are a cartesian product of the matching tuples in the joins of each subtree under R_i and the join of the remaining relations.

This implies both that the counts computed in the counting phase are correct, and that the enumeration phase forms the correct output.

5.5 Complexity

We account for the work done by the algorithm that is not captured by the $O(t_{\text{reduce}})$ term:

- The work spent on the counts in the counting phase.
- The work spent on the intervals in the enumeration phase.

Note that by acyclicity, and since no set of attributes is contained in another, it follows that each relation has at least one unique attribute. First consider the counting phase: Any count can be matched uniquely to an attribute value in the output (that of the unique attribute of the tuple), and each count is part of the input to a constant number of sorting steps, so the added complexity is $O(\text{sort}(z))$. In the enumeration phase, the number of intervals is bounded by the number of attribute values in the output. Thus, the total size of the encoding of all intervals is at most z blocks. The cost of handling counts in this phase is the same as in the counting phase. In conclusion, for both passes the total size of the input to all sorting algorithms is $O(z)$ blocks, plus the amount of data sorted in the full reducer (times some constant). The complexity stated in Theorem 1 follows.

Consider the join of the following relation instances:

A	B
1	22
2	99
3	55
4	55
5	66

R_1

B	C
22	111
22	888
55	222
55	333
66	777

R_2

C	D
111	a
222	c
222	e
333	d
888	b

R_3

(The relations are not fully reduced, but this does not affect the correctness of the algorithm.)
We choose R_1 as root and R_3 as leaf of the tree R_1–R_2–R_3. The counting phase computes occurrence counts for tuples in R_3, R_2, and R_1 (in that order) as follows:

A	B	c_{R_1}
1	22	2
2	99	0
3	55	3
4	55	3
5	66	0

B	C	c_{R_2}
22	111	1
22	888	1
55	222	2
55	333	1
66	777	0

C	D	c_{R_3}
111	a	1
222	c	1
222	e	1
333	d	1
888	b	1

Then the prefix sums of the counts of R_1 are computed, and we associate an interval of row numbers in the final join result with each tuple:

A	B	i_{R_1}
1	22	[1;2]
2	99	$\emptyset$
3	55	[3;5]
4	55	[6;8]
5	66	$\emptyset$

From the intervals on the tuples of R_1 we can compute the intervals for R_2 by joining on **B**:

B	C	i_{R_2}
22	111	[1;1]
22	888	[2;2]
55	222	[3;4], [6;7]
55	333	[5;5], [8;8]
66	777	$\emptyset$

From the intervals of R_2, and with *no need for the information on* R_1, we can compute the positions for tuples in R_3 by joining on **C**:

C	D	i_{R_3}
111	a	[1;1]
222	c	[3;3], [6;6]
222	e	[4;4], [7;7]
333	d	[5;5], [8;8]
888	b	[2;2]

The row number intervals associated with all tuples allow us to form the join result: First split intervals with more than one row number into "intervals" of one row, then sort according to row number (keeping track of which relation each tuple belongs to). A single scan then gives the result tuples.

Figure 1: Example run of the acyclic join algorithm.

5.6 Handling small domains

In this section we sketch the main ideas that go into extending our results to hold regardless of the attributes' domain sizes. The main technique is to keep the representation size of the various numbers (tuple references, intervals, and counts) down by efficient encoding. For example, counts in the counting phase should be coded in an efficient prefix code such that a count of x is encoded in $O(\log(x+2))$ bits. The interval boundaries in the enumeration phase should be *difference coded*, i.e., each number should be encoded as the difference from the previous number, using an efficient prefix code for the integers. Since the intervals appear in sorted order, this reduces the size of the representation of intervals to $O(k)$ bits per result tuple. Finally, when performing joins over attributes from a small domain the associated row numbers may dominate the complexity. Again, the resort is to maintain row numbers in sorted order and difference coded (using, e.g., multi-way mergesort). This keeps the cost of performing the join down to a constant factor times the cost of sorting the common attributes.

6. 3-CYCLE JOIN GRAPHS

In this section we sketch our algorithm for computing the join of three relations whose join graph is a 3-cycle, i.e., where any pair has a common attribute not shared by the third relation. In this case there are examples in which the join of any two relations has quadratic size, even though there are *no* tuples in the final result. Consider for example three relations with schemas `(a,b)`, `(b,c)`, and `(a,c)`, each containing the following set of $2N$ tuples:

$$\{(0,i),\ (i,0) \mid i = 1,2,\ldots,N\}\ .$$

There are no dangling tuples, and the join of any two of the relations has N^2 tuples.

Our algorithm starts by sorting each pair of relations according to their common attributes with each of the other relations. To each possible value on the common attributes we can then associate a unique integer. This allows us to reduce the problem to the case of relations R_1, R_2, R_3 with schemas `(a,b)`, `(b,c)`, and `(a,c)`, respectively. We make sure that the same value is not used for two distinct attributes.

The basic idea is to associate with each distinct attribute value x a random number h_x from some large range $\{1,\ldots,L\}$. This is done using sorting. To a tuple $t \in R_1$ we associate the value $h_{t(\mathtt{a})} - h_{t(\mathtt{b})}$, and similarly to a tuple $t \in R_2$ we associate the value $h_{t(\mathtt{b})} - h_{t(\mathtt{c})}$. Finally, to a tuple $t \in R_3$ we associate the value $h_{t(\mathtt{c})} - h_{t(\mathtt{a})}$. Observe that for any tuple in the result, being a combination of $t_1 \in R_1$, $t_2 \in R_2$, and $t_3 \in R_3$, the values associated sum to zero. Conversely, for any three tuples $t_1 \in R_1$, $t_2 \in R_2$, and $t_3 \in R_3$ that do *not* match, the probability that their associated values have a sum of zero is at most $1/L$. Thus, if L is chosen large enough, we have that with high probability the result tuples correspond exactly to the triples of associated values having sum 0. These triples can be found in $O(n^2/m+\mathrm{sort}(n))$ I/Os by a recent result of Baran et al. [2]. Once the triples have been found, we remove the tuples in each relation that is not in any triple, i.e., not part of any output tuple, and perform the join using two binary joins. This uses $O(\mathrm{sort}(z))$ I/Os, since the intermediate result has size $O(z)$ blocks, with high probability.

7. CONCLUSION AND OPEN PROBLEMS

Our main result is a worst-case efficient external memory algorithm for computing k-ary joins, whose complexity does not grow with k. Our algorithm is also much more *predictable* than previous methods: Assuming that the result of the join is no larger than the input relations, a good complexity bound for a concrete join can be computed from the amount of data in each attribute of each relation.

In addition to having good worst-case behavior, we believe that our algorithm is more efficient than previous algorithms for typical joins involving many relations. It would be interesting if this was investigated experimentally. In general, our algorithm (or a more practical variant of it) could be integrated into a normal query optimizer architecture by evaluating execution plans in which the possibility of executing a join of (part of) the relations using our algorithm is taken into account.

From a theoretical perspective it would be interesting to try to extend the results to more general relational algebra expressions, e.g., a join followed by a projection. Finally, it is an intriguing open problem whether our result for 3-cycles can be extended to k-cycles, for $k > 3$.

Acknowledgements. The authors would like to thank Gerth Stølting Brodal and Uri Zwick for useful discussions related to this paper.

8. REFERENCES

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Comm. ACM*, 31(9):1116–1127, 1988.

[2] I. Baran, E. D. Demaine, and M. Patrascu. Subquadratic algorithms for 3SUM. In *Proceedings of WADS*, volume 3608 of *Lecture Notes in Computer Science*, pages 409–421, 2005.

[3] E. F. Codd. A relational model of data for large shared data banks. *Comm. of the ACM*, 13(6):377–387, 1970.

[4] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–72. ACM Press, 2002.

[5] J. M. Hellerstein and M. Stonebraker. *Readings in Database Systems*. MIT Press, 4th edition, 2005.

[6] T. Ibaraki and T. Kameda. On the optimal nesting for computing N-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, 1984.

[7] Y. E. Ioannidis. Query optimization. In *Computer Science Handbook, Second Edition*, chapter 55. Chapman & Hall/CRC, 2004.

[8] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 268–277, 1991.

[9] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 168–177. ACM Press, 1991.

[10] D. Maier, Y. Sagiv, and M. Yannakakis. On the complexity of testing implications of functional and join dependencies. *J. Assoc. Comput. Mach.*, 28(4):680–695, 1981.

[11] S. Pramanik and D. Vineyard. Optimizing join queries in distributed databases. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 287 of *Lecture Notes in Computer Science*, pages 282–304. Springer, 1987.

[12] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–34. ACM Press, 1979.

[13] J. D. Ullman. *Principles of Database and Knowledge-based Systems*, volume 2. Computer Science Press, 1989.

[14] D. E. Willard. An algorithm for handling many relational calculus queries efficiently. *J. Comput. System Sci.*, 65(2):295–331, 2002.

[15] M. Yannakakis. Algorithms for acyclic database schemes. In *7th International Conference on Very Large Data Bases (VLDB)*, pages 82–94. IEEE, 1981.

[16] C. T. Yu and M. Z. Ozsoyoglu. An algorithm for tree-query membership of a distributed query. In *Proceedings of Computer Software and Applications Conference*, pages 306–312. IEEE, 1979.