Report

Reducing duplicate work in relational join(s): a unified approach

Author(s):
Rosenthal, Arnon; Rich, Christian; Scholl, Marc H.

Publication Date:
1992-01

Permanent Link:
https://doi.org/10.3929/ethz-a-000628397

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
Reducing Duplicate Work in Relational Join(s): A Unified Approach

January 1992
Reducing Duplicate Work in Relational Join(s): A Unified Approach


* MITRE Corp., Bedford MA; work done while visiting ETH Zürich
** Dept. of Computer Science, ETH Zürich
   e-mail: arnie@mitre.org, {rich,scholl}@inf.ethz.ch

Abstract

Most join algorithms can be extended to reduce wasted work when several tuples contain the same value of the join attribute. We show that separating detection of duplicates from their exploitation improves modularity and makes it easier to implement whole families of hierarchy-exploiting join algorithms that avoid duplication.

The technique is also used to provide an execution technique for star-like patterns of joins around a central relation. It appears to dominate Ingres-like substitution for the central relation, in both performance and ease of including in a conventional optimizer. Its performance dominates a cascade of conventional binary joins, and performance estimates are more accurate.

1. Introduction

Due to normalization, which strips down data to many tables to reduce redundancy, the join operation is used quite often in relational queries. When object-oriented database systems - supporting enhanced complex applications - use relational systems for storage, reassembling the objects to query on them will even increase the number of expected joins per query. Further, mostly the join has the lion's part of the query cost. Due to that, a significant amount of effort had been and will be spend on developing efficient join algorithms.

There are many proposals for computing joins [BE76, DKO*84, Val87, Omi89, SC90, Thy90, Gra91]. Most commercial relational systems support Nested Loops and Sort Merge Join. Nested Loops exploits index on the join attribute, and is often the best method when the join predicate has a high selectivity [BE76, Ull89]. Sort merge makes use of bulk sequential I/O's and is often the best, when the number of qualified tuples from both relations is high and the join predicate does not offer much filtering. Hash Join [DKO*84, DGS88] has proved to be fast, especially when one relation nearly fits in main memory, in addition it is easy to parallelize. Depending on the situation, the performance of different join strategies can vary by magnitudes. It is the optimizers task to choose the appropriate one, if more than one join strategy is supported for a given task.
In spite of the good performance presented for these join algorithms in the literature, most join algorithms do unnecessary work, when several tuples of a join's outer relation contain the same value of the join attribute. (Index-) Nested Loops can access both pages and tuples repeatedly. Sort Merge Join and Hash Join try to avoid redundant access to pages, but are still redundant at the tuple level. We present an algebraic approach to exploiting this duplication on a tuple level. The key ideas for implementing a single join are 1) the duplication to be exploited is identified in one module, and the actual join is performed in a separate module; and 2) the results of the identification phase are represented as a nested (NF²) relation [SS86] that groups the identified duplicates. This relation can be input to join and other operators. The major benefit of this approach is that it provides a very flexible, modular approach for reducing duplications with a wide variety of duplicate-detection mechanisms and join implementations.

For simplicity, we focus on (Index-) Nested Loops Join. This algorithm obtains the greatest savings because we reduce duplicate page access, while Sort Merge and Hash Join save only on duplicating in-memory tuple manipulations. The savings can still be significant, since NL-join is frequently the selected strategy. In fact, for some joins, (index-) nested loops is the only strategy supporting the given task. For example, most non-equijoins, or joins with external join predicates¹ have no other implementations. On the other hand, the most frequent joins, i.e., natural joins, are typically supported by (one or two) indices. In this case, Index Nested Loops is the strategy of choice.

Our second result is to extend our approach to obtain a unified implementation for multiple joins in star queries of k≥2 joins. As asserted in [WY76, KS86, UI89, Cha91], a multiway join algorithm (like Ingres tuple-substitution) is frequently better than any sequence of binary joins. We show that a cascade of binary hierarchical joins (HJOIN) combines the advantages of two competitors: a cascade of binary joins using (index-) nested loops, and Ingres tuple substitution [SWK*76, WY76]. It will also require lower implementation effort than adding a multiway (k-ary) join to the optimizer's target machine. More generally, any (binary) join plan containing at least one (Index-) Nested Loops Join can be improved by converting this to an HJOIN. Furthermore, we show that HJOIN's are less sensitive to erroneous estimates of join selectivity. We therefore conclude that an optimizer's abstract target machine should not include a k-ary join operator.

The paper is organized as follows: Section 2 introduces the idea of reducing duplicate work by "hierarchicalizing" single joins and compares with other possible tactics. In Section 3, we apply the approach to queries with multiple joins and give qualitative and quantitative analyses. Experiments that verify the theoretical results are presented in Section 4, before we conclude in Section 5.

¹ e.g., joins in geoscientific applications, testing the "inside" property of cities and a region.

2. Technique for 2-Relation Joins

2.1 Potential Savings from Duplicates, and How to Exploit Them

For most join implementations, there is an outer² and an inner relation, and for each tuple of the outer relation, it is necessary to execute "Fetch tuples of inner relation that correspond to current tuple of the outer relation". Within a sequence of tuples with the same join attribute value (called a run), the fetched tuples are identical.

For example, consider the join of relations R₁ and R₂ on predicate p(A) = "R₁.A = R₂.A". A straightforward nested loop join, (using indexes if available) accesses relation R₂ (index IR₂.A plus the resulting pages, respectively) four times, once for each R₁ tuple. This wastes effort, because the matching R₂ tuples fetched for the outer relation tuples <2,aa>, <2,bb> and <2,dd> will be identical (<2,44> and <2,II>). Suppose instead we grouped R₂ into a nested relation R₂*: = \{A, B_{R₂}(R₂)\} in which all B values of tuples that have the same value for attribute A are grouped together into a "subrelation" B_{R₂}(R₂). R₁' join R₂' will then require only two accesses to relation R₂ (index IR₂.A plus the resulting pages, respectively) because there is only one access on R₂ (Index IR₂.A plus the resulting pages, respectively) for every tuple of R₁'.

This hierarchicalized join plan is shown algebraically in the operator tree below the example. It consists of three operations, which may be pipelined. The first phase (a group operator, γ) reduces the number of tuples by grouping on the join attributes (and possibly others), producing a nested relation where all join attributes are in the root. The second phase (the join operator) joins the nested relation with the join's inner operand. The combination of these two steps, grouping and joining, is called HJOIN. The third phase (an ungroup operator, δ) flattens the join result, if so requested. Notice that it will often suffice to flatten only at the very top of the operator tree, immediately before delivering the final result.

We use non-first-normal-form (NF²) relations [SS86] as the result of the identification phase. In NF² relations, attributes may either be atomic-valued or relation-valued. NF² relations can be derived from flat relations by simply applying the concept of a relation in a nested manner, thus resulting in a hierarchical data structure. The motivation for using a nested rather than a flat representation is that the nested relation often uses fewer tuples (especially) and also fewer bytes to represent the same information. In particular, a costly computation (e.g., "retrieve corresponding tuples of the 'inner' operand") that needs to be

² Other names are sometimes used (e.g., building and probing relations for hash join)
done for each value of a non-repeating attribute will be invoked fewer times for a stream of nested tuples.

Our approach separates identification of duplication from its exploitation, in a modular fashion. This is important, as it makes it unnecessary to reimplement every join-like operator, along three dimensions: 1) the operation’s output (is it join, outerjoin, semijoin, …); 2) implementation tactic (e.g., nested loop, merging, partitioning, …) and 3) thoroughness of removing duplication (section 2.2). Our ideas seem to apply to all of these cases, as long as the operators can handle tuples with varying-length attributes (which hold the nested relations). Particularly, relational join is still defined for nested relations — the definition works regardless of whether attributes are atomic (integer strings), relations, or variable-length byte-strings, or whatever (maps, documents). In fact, we use only atomic attributes in join predicates, so subrelations are only copied as a whole.

2.2 Duplicate-Identification Phase

We now describe how a relation (viewed as a stream of tuples) can be represented (with less duplication) as a stream of $N^2$ tuples. The thoroughness (and cost) of duplicate elimination can be selected by the optimizer, and adjusted modularly, depending on the anticipated savings in the actual joining, and on the desire to preserve the current ordering of the tuple stream.

Processing is assumed to be based on pipelining and dataflow query processing techniques [SC75, BD80, Gra90]. Data elements are passed between two operators through a stream, that is a consumer-producer queue, comparable to a Unix pipe. A streamed relation is therefore a sequence of tuples rather than a set.

Generically, $\gamma[A, B_{set}](R)$ denotes a relation grouped on attributes of the set $A$. The other attributes of $R$, say $B = atr(R) - A$ are nested into a new subrelation (relation-valued attribute) named $B_{set}$. That is, for each group of $A$ values we get a tuple, and the set of $B$ values leads to a set of sub-tuples. We define a family of grouping operators, differing in the extent to which they aggregate tuples that agree on the $A$ attributes, ranging from removal of all duplication (called value-based grouping), through removals achieved when adjacent tuples have the same $A$-value (called run-based grouping) to no removal at all.

Value-based grouping forms a single group for each value of the root attributes. It removes the maximum amount of duplication, potentially at high cost, and may be implemented by sorting or hashing to find duplicates.

Definition: (value-based group)

$\gamma[A, B_{set}](R)$ produces one $N^2$ tuple for each distinct value of $\pi[A](R)$.

(Notice, that value-based grouping is actually the nest operator of [SS86]: $\gamma[A, B_{set}](R) = \gamma_{B_{set}=\text{nest}(R)-A}(R)$)

$\gamma$ can be expensive, but it is an appropriate choice if reducing duplicates is very important (e.g. multirelation join, in Section 3), or if there are other reasons to sort or group on $A$.

Other times it may be appropriate to use a grouping operation that removes only duplicates that can be identified cheaply. An example, run-based grouping, is discussed below.

Definition: A run [of attributes $A$ in streamed relation $R$] is a maximal sequence of consecutive tuples that have the same value for $\pi[A](R)$.

Runs can arise in many ways, in practice. For example, sometimes we join on the field that determines physical placement, e.g., Employees might be clustered by Department. (Newly inserted tuples may sometimes appear elsewhere, somewhat increasing the number of runs produced by a physically-sequential scan). As another example, the attribute used for physical clustering may correlate with the join field; for example, if document_ids are created sequentially, then a relation ordered by document_id may be ordered by creation_date and have substantial runs for completion_date. When several joins are cascaded, each join’s output will contain runs of tuples that correspond to the same outer-relation tuple (see below).

Runs can be exploited to cheaply do groupings. In addition to being cheaper, run-based grouping (denoted $\gamma$) preserves the sort ordering of its input, an ordering that may be desired by the user or by a subsequent merge join. We now give operators to group and ungroup a relation based on runs.

Definition: (run-based group)

$\gamma[A, B_{set}](R)$ produces one $N^2$ tuple for each run of $A$ in stream $R$.

Definition: (run-based ungroup)

$\gamma[B_{set}](R)$ is inverse to run-based group. It also touches each tuple just once, does not check for duplicates, and is very fast. Value-based ungrouping is equivalent to the unnest operator of [SS86].

Example:

Consider the streamed relation $R$:

\[\begin{array}{ccc|ccc|c|c}
R & A & B & R & A & B_{set} & R & A & B_{set} \\
--- & --- & --- & --- & --- & --- & --- & --- & --- \\
2 & aa & 2 & bb & 2 & (aa,bb) & 2 & (aa,bb) \ \\
3 & cc & 3 & cc & 3 & (cc) & 3 & (cc) \\
2 & dd & 2 & dd & 2 & (dd) & 2 & (dd) \\
\end{array}\]

Note, that since SQL systems already have a value-based grouping operation (GROUP BY), relatively little extra implementation effort is needed.

2.3 Duplicate-Exploitation Phase — Hierarchicalized Join

A hierarchicalized join consists of 1) a group operation, followed by 2) a join operation. When relational output is needed (e.g., at the conclusion of all joins), an ungroup is performed.

The grouping operation is selected by the query optimizer, e.g., none, run-based, or value-based, depending on cost estimates. It will group by at least the attributes that appear in
the join predicate. (Attributes functionally dependent on the join attributes may also be included in the root, as this will not increase the number of groups, and will avoid replicating their values in each nested subtree). In the example above, \( B \) was not functionally dependent on the join attribute \( A \), therefore, it was nested. The benefit is that expensive operations using the root attributes (as starting subqueries on the inner relation) are performed only once per group of outer tuples.

Now assume that the next operator following the join is another join (on the same attribute(s) of the outer relation). The first join produces runs of tuples that agree on all the outer relation’s attributes. Therefore, another — run-based (!) — group operator between the two joins will reduce the amount of work for the second join. We will elaborate on this observation in Section 3.

2.4 Other Techniques for Exploiting Duplicates — What’s Wrong With Them

Most relational systems exploit duplicate values only as a by-product of caching. That is, the buffer manager checks whether a page it is fetching is still in memory from a previous fetch. But even if the page need not be fetched, considerable effort is duplicated for indexing, address translations, and the like; worse yet, in many cases the needed pages may have been swapped out.

To avoid this waste, duplicates could be detected earlier, based on examination of tuples rather than page references. One way is for the join code to look-ahead at the next tuple (i.e., to detect runs). But the look-ahead requires accessing additional tuples (which might not be needed if the query is halted before all satisfying tuples are returned). Also, look-ahead code must be inserted into every join variant (e.g., outerjoin, semijoin, etc.) and every implementation variant (nested loop, hash join, etc.). Finally, this run-detection gives no help if one really wants a more thorough form of duplicate detection.

A third approach, called lazy pipelining in \([CGK90]\), is to remember the results of all previous joins in main memory. A variant that remembers only whether a tuple failed to find a match requires just one bit per outer-relation tuple. Again, this technique requires modifying each one of the implemented physical join operators; furthermore, it interacts in subtle ways with other speedup techniques, such as intelligent backtracking \([CGK90]\).

3. Techniques for Star-Queries

Joins involving more than two relations are common today and will be more common tomorrow, due to OODBMS’s and KBMS’s that use relational databases for storage. In OODBMS’s, the decomposition of “complex objects”, where one has to normalize several object-valued properties of objects into several 1NF relations is a typical situation. These relations are “connected” via key – foreign key relationships on the same attribute (set).

Reconstruction of the application object requires a (star) join query. Techniques for processing these joins efficiently may gain in importance.

This section first shows how our technique may be applied to any cascade of binary joins; often it will yield an improvement. Next, we examine Ingres tuple substitution — a kind of k-ary join operator — and compare it with a cascade of (index-) nested-loop joins, and with the “hierarchicalized” form of multiple (index-) nested loops, here denoted HJOIN.

We examine situations where each performs well, and show that HJOIN dominates the other two. Finally, we show that hierarchicalized join plans are less sensitive to errors in selectivity estimation.

We present the result for “Star” queries — that is Select/Project/Join queries such that the center relation (denoted \( R_c \)) appears in every join predicate. The discussion also applies to processing the star subgraphs of more general queries.

3.1 Hierarchicalizing Existing Implementation Algorithms for Multirelation Joins

In this section, we show how a join algorithm generated by an ordinary query processor might be hierarchicalized.

All common join implementations produce runs of tuples in the join result that agree on all attributes of the outer relation. This is obvious for nested loops, where we find all matching inner tuples for an outer tuple before considering the next outer tuple. It is, however, also true for sort-merge join and hash-based join algorithms. In hash-based algorithms, runs are on the attributes of the “probing” relation. Therefore, execution plans for multiple-join queries can easily be hierarchicalized by inserting (run-based) group operators between any two join nodes to group by the outer relation’s attributes.

Let \( R_c, jn_1, R_1, jn_2, R_2, \ldots, jn_p, R_p \) denote a sequence of join operations, where \( jn_q \) denotes some implementation of the \( q \)th join. The fully hierarchicalized form is obtained by grouping the complement of the outer relation’s attributes into a subrelation after each join, and flattening the final result. That is, the hierarchicalized form is:

\[
\text{stream}_1 = \gamma R_c [R_1 \ldots (R_p jn_1 R_1)]; \\
\ldots \text{stream}_i = \gamma R_c [R_1 \ldots (\text{stream}_{i-1} jn_i R_i)]; \\
\ldots \text{result} = \text{union \ldots (\text{result} [\text{stream}_p])}
\]

The first join in this sequence might also be hierarchicalized, e.g., if \( R_1 \) contains runs on the join attributes. Further improvement can be obtained by writing a single routine that joins and produces grouped results directly. While this is less modular than performing run-based group after the join, the added efficiency may be worthwhile in the most common cases (e.g., hierarchicalized nested loops).

Hierarchicalizing tends to reduce the cost of (index-) nested-loop steps, by reducing the number of duplicate page accesses. Actually, it is possible to obtain hierarchical forms of
most join implementations, not just nested loops. Most implementations have a step where, for an outer tuple, one a) determines the corresponding set of inner tuples; and 2) produces the joined output corresponding to those inner tuples. The hierarchical form of any join algorithm simply produces nested tuples, rather than multiple flat ones. Hierarchicalizing tends to have smaller effects on the cost of some other implementations, such as sort-merge and hashed-join. Fewer, but larger tuples will have to be sorted/hashed. The total amount of memory needed, however, will be reduced due to duplicate elimination for root attributes.

As already mentioned, there may be some joins, which cannot be implemented other than by (index-) nested loops (for example, nonequi-joins or joins with external join predicates). For queries with a large number of joins, an execution plan might contain a variety of join implementations. In this plan, the cost is easily hierarchicalized by introducing a group operator before each (Index-) Nested Loops Join. In contrast, utilizing a multiway join operator would mean to completely reorganize the execution plan. Even in this case, the savings earned through hierarchicalizing can be substantial. The same is true for any other costly operation performed on the grouped attributes, which then had to be done just once for each group of tuples.

3.2 Comparison of Three Join Plans for Queries with Two Joins

We now present three algorithms cascaded nested loops, a multiway join (Ingres-like), and hierarchicalized cascade of nested loops and compare their performance and trade-offs. All three algorithms are assumed to use indexes, if available.

Notation: For simplicity, our examples show only three-join relations. Assume that $R_1 \Join \ldots \Join R_n$. We associate left to right but use $R_n$ as the outer relation of all joins. The generalization to $k$ relations is straightforward. $\text{Card}(R)$ denotes the number of tuples in $R$.

a) Cascade of binary (Indexe-) Joins. A cascaded nested loops plan (using indexes) performs one "Find matching tuples in first inner relation" for each tuple of the center relation $R_c$, and one (on the second inner relation) for each tuple of $(R_c \Join R_2)$. That is, it can be described by the following cost expressions.

- for the first join (NL):
  $\text{Card}(R_1) \times \text{Cost (subquery on $R_1$)}$
- for the second join (NL):
  $\text{Card}(R_c \times R_2) \times \text{Cost (subquery on $R_2$)}$

The cost of the second join is driven by the factor $\text{Card}(R_c \times R_2)$, which can be substantial. However, the matching tuples found in relation $R_2$ are determined by the corresponding $R_c$ tuples. We next show how a $k$-ary join operator exploits this fact, by taking both joins at once.

b) A multiway Join Operator. A multiway join plan performs for each tuple of the center relation one "Find matching tuples in first and second inner relations". Its sequence of operations cannot be simulated by a sequence of 2-way joins and is frequently more efficient than any sequence of 2-operand joins on flat tuples. The Wong–Youssefi algorithm, implemented in the Ingres research prototype, included a tuple substitution operation [WY76, YW79] that is algebraically a multiway join. There have been (at least informal) proposals to include multiway join within an optimizer's repertoire [WY76, KS86, Ull91]. The multiway join (used in Ingres) is:

For each $t_c$ in $R_c$:
- $S_{c1} = \text{Find} (t_1 \mid t_1 \in R_1, P_{c1}(t_c, t_1))$ /* retrieve from $R_1$, store in $S_{c1}$ */
- $S_{c2} = \text{Find} (t_2 \mid t_2 \in R_2, P_{c2}(t_c, t_2))$ /* retrieve from $R_2$, store in $S_{c2}$ */

Result = $\text{Result} \cup (t_c \times S_{c1} \times S_{c2})$

This algorithm starts subqueries against both inner relations for each tuple of the center relation, that is, its costs are:

- for the first join (NL): $\text{Card}(R_1) \times \text{Cost (subquery on $R_1$)}$
- for the second join (NL): $\text{Card}(R_c) \times \text{Cost (subquery on $R_c$)}$
- final "local product": $\text{Cost of combining $R_1$ and $R_c$-matches}$

Multiway join appears more difficult to add to a query processor (see section 3.4), than a binary join operator. We now show how to exploit duplicates by using a binary join operator.

c) Cascade of HJOIN – Our Multiple Join Algorithm. A hierarchicalized cascade of nested loops will contain a run-based grouping operation between the first and the second join, therefore, the second join starts only one subquery on $R_2$ per $R_1$-tuple that had a match in $R_1$. This number is the cardinality of the semi-join between $R_c$ and $R_1$. The cost expression is:
3.3 How Cascaded HJoins Reduce the Dependence on Database Statistics

Estimating the size of a multirelation join is notoriously difficult [IC91]. Yet erroneous size estimates can cause selection of a bad execution plan. Proposed solutions (e.g., sampling) require considerable implementation effort and run-time overhead. The difficulty is most noticeable when running time for a cascade of relational joins is dominated by the last join, where one operand may be an intermediate that is much larger than any of the input relations.

The Cascading HJOINS approach reduces the dependence of the optimizer on estimates of the sizes of intermediate results. If each join increases result size, then the last (and least accurately analyzable) join dominates running time. With cascaded HJOINS (and multiway join), the number of joining tuples at each stage is at most the number in the original outermost relation. So accurate estimates of the number of join selectivity (here reflected in the count of subtuples) are less critical.

Consider for example a sequence of joins, such that the estimated growth factor of one join is off by a factor of two. This results in an erroneous result size estimation by a factor two, for all the following joins. This effect gets even more dramatic if several joins were estimated wrongly. For example, if two join selectivity estimates were off by a factor of two, this resulted in a total error of a factor of four, since the errors multiply. Using the HJOIN strategy, due to the grouping of matching inner relation tuples into one subrelation, the number of result tuples will always be less than or equal to the number of tuples of the outer relation, and therefore differ only slightly from the estimate. Errors add but do not multiply, since the combination of matching tuples from different inner relations is done only in the very last step, the ungroup. For shrinking joins, the effect is less dramatic.

3.4 How to Add Hierarchicalizing to a Conventional Query Processor

We have seen two alternatives for improving performance on multirelation joins - to add hierarchicalizing (particularly for (index-) nested loops) or to add a k-ary join operator. In this section we show how the cascade of HJOIN's algorithm can be incorporated rather easily, while a k-ary join operator introduces undesirable complication. Furthermore, a cascaded HJOIN allows free choice of implementation for each join, while the Ingres k-ary join (by substitution) appears to mandate nested loop approach to each of the joins.

Implementation as a series of 2-operand joins fits easily into the normal modularization of a query optimizer. We assume a conventional bottom-up dynamic program for strategy generation, as in System R and related work [SGA79]. The optimizer generates join orderings for an entire Select-Project-Join query block. For each proposed join, it examines a sequence of implementations. HJOIN can be simply added to the list of possible implementations of each join, whenever the query graph includes a star subquery. Just as some join implementations may need to be succeeded by Sort to obtain a desirable order, HJOIN may need to be succeeded by ungroup, to obtain the desired physical property "flatness".

To use HJOIN, one need implement only a very limited form of nesting (one level), plus the group and ungroup operators (some form of grouping typically has already been implemented for SQL-"GROUP BY"). We assume also that relational joins have already been implemented in a way that permits variable-length attributes, at least in non-join fields.

If the joins in the star subquery are all on the same attribute, one grouping suffices. If not, we either need several groupings/ungroupings, or have to group on all join attributes at once.
To handle multiway join, we face at least the following problems:

- **Identifying desirable k-ary joins from within larger queries.** At the least, one would need to rewrite the module that takes a query block and (bottom-up) generates expressions in terms of two-way joins.

- **A plethora of logical-level forms of k-ary joins** must be implemented. These present numerous difficulties, e.g., how to handle an expression like \( R_1 \cap R_2 \) outerjoin \( R_3 \)? In a cascade of binary operators, implementations can be chosen individually.

- **A plethora of physical implementations of k-ary joins** must be implemented. Again, a cascade of binary joins handles the problem, while for k-ary joins it is not clear where the decision would be made. For example, where in a k-ary join should the optimizer consider sorting a relation before a join? What does each option do to secondary sort keys?

- **Complication in the basic execution model** – k-ary join requires support for a variable number of arguments to operators, but the target language for execution may not support such operations.

So introducing a multiway join operator seems much more severe than defining a new join implementation and nesting as a new physical property.

### 4. Performance Measurements

In this section, we report on an experimental evaluation of the performance and behavior of HJOIN, compared to (Indexed-) Nested Loop join and Multiway join.

The Join algorithms are implemented in C, on top of the page layer (Stable Memory Manager, SMM) of the DASDBS [PSS*87] database kernel system. The experiments were run on a Sun 3/80, without other users. The machine had 8 megabytes of main memory, however the data base buffer pool was chosen small enough, such that neither relation fits in. We focus on a three relation join, although the results will be even better for joining more relations.

The relation sizes we chose for the experiments may appear too small (5 to 100 pages). But first of all this size was enough to show the effects, and second, typical applications will not ask for complete joins between 8 relations. Rather, one will have selections in the query to restrict the attention to some small part of each involved relation. Also, some of these selections will be supported by an index. Since we did not want to implement indices also, the restriction to rather small relation sizes emulates previous index evaluations, that is, the number of tuples qualified in the index look up.

#### 4.1. Experiment 1: Cost increase for processing nested, instead of flat tuples

Using HJOIN instead of Nested Loop will decrease processing cost, especially in the case of multiple joins. Nevertheless, processing on nested tuples, grouping and ungrouping of tuples, which is needed in the case of HJOIN, produces some CPU overhead. To see, how expensive this overhead becomes, we ran the following experiment:

We join two relations, \( R \) and \( S \), with a primary to foreign key join predicate. The relation sizes of \( R \) and \( S \) are equal, and are varied from 5 to 100 pages. We measure the response time, processing this join with Nested Loop join and HJOIN.

In this experiment we study the sensitivity of required I/O's to the relation sizes. We consider the query \( R_1 \text{[pc]} R_C \text{[pc]} R_2 \), in a primary to foreign key manner. \( R_C \) is the center relation, which is the outer relation for the two joins. All three relation sizes are equal, and are varied in the range of 5 to 100 pages. The join column values in relation \( R_C \) are unique, while the ones in \( R_1 \) and \( R_2 \) are drawn from a 80–20 Zipf distribution, over the active domain of the \( R_C \) join column. That is, 80% of the join column values are drawn from 20% of the \( R_C \) join column domain. We measure the number of logical I/O's produced, that is the number of DASDBS Stable Memory Manager buffer fixes, and
the resulting response time, for processing the query with 1) Nested Loop join, 2) Ingres multiway join and 3) HJOIN.

This is because the result size of $R_c$ tuples will increase, and therefore raise the number of outer tuples in the second join. This leads to a quadratic cost function. Using Ingres Multiway join, the number of outer tuples in the second join is constant, equal to the relation cardinality of $R_c$. This leads to a linear cost function. HJOIN is again better, due to the fact, that $R_c \mid R_1$ contains only tuples of $R_c$, that have matching $R_1$ tuples.

Figure 4.2: Three Relation Join

Figure 4.2 shows that the number of I/O's for Nested Loop join and Ingres multiway join is exactly the same. This is particular for our example relations: because the number of resulting tuples produced by joining $R_c \mid X \mid R_1$ is the number of $R_1$ tuples, and since the relation sizes are equal, this corresponds to the number of $R_c$ tuples. HJOIN needs less I/O's, since the number of tuples produced by joining $R_c \mid X \mid R_1$ is less, due to the grouping of matching $R_1$ tuples into the corresponding $R_c$ tuples. In this experiment, we get an average of 3.2 subtuples, which reduces the work to be done in the second join to 31% on average, of that from Nested Loop join or Ingres multiway join. For processing both joins, the total number of page accesses, and the response time (which is mainly determined by the number of page I/O's) decreases on average to 65% and 67% respectively. That is, HJOIN outperforms Nested Loop join and Ingres multiway join concerning response time, on average by 33%.

4.3 Experiment 2.2

In this experiment, we study the effect of having different relation sizes. We changed the experimental structure of experiment 2.1 in the sense, that we keep the relation size of $R_c$ fixed. The relation sizes of relation $R_1$ and $R_2$ are still varied in the range of 5 to 100 pages.

In Figure 4.3 we see, that Nested Loop join gets worse, for increasing inner relation sizes. This is because the result size of $R_c \mid X \mid R_1$ will increase, and therefore raise the number of outer tuples in the second join. This leads to a quadratic cost function. Using Ingres Multiway join, the number of outer tuples in the second join is constant, equal to the relation cardinality of $R_c$. This leads to a linear cost function. HJOIN is again better, due to the fact, that $R_c \mid X \mid R_1$ contains only tuples of $R_c$, that have matching $R_1$ tuples.

4.4 Experiment 3: Varying the Join Selectivity

In this experiment, we study the sensitivity of required I/O's and response time to the join selectivity.

We consider the query $R_1 \Join \left[ p_{c1} \right] R_c \Join \left[ p_{c2} \right] R_2$, keeping the relation sizes fixed and varying the join selectivity. We draw the join column values from an equal distribution, over the join column domain size, which is varied according to the desired join selectivity.

For a better understanding, we choose the growth factor as the unit over which the join selectivity does not influence the number of logical I/O's and the response time at all. Further, Nested Loop join is superior to Multiway join for a decreasing join (growth factor < 1), but
gets worse for a increasing join. HJOIN, combining the advantages of Nested Loop join and Multiway join, is superior to both, in terms of I/O cost and response time.

5. Summary and Conclusions

We investigated tactics to avoid duplicate accesses to the same tuples, for single and multiple joins. Our approach is to apply a duplicate–combination function (grouping) to produce a relation nested on the join attributes, and then to join the nested relation. Thus, we have provided a unified treatment that dominates numerous ad hoc techniques. Furthermore, the approach can be implemented very modularly, without modifying numerous join implementations, such that grouping can be inserted into all desirable join variants. The resulting architecture is unified for every costly operation (like join) performed on the grouped attributes. Other proposed approaches (e.g., lazy pipelining [CGK90]) can be approximated as particular duplicate–combination functions. We also provide flexibility in degree of effort, i.e., the ability to do either full or partial grouping to remove duplication. The technique fits well with conventional optimizers — the greater generality does not appear to cause increases in implementation effort or run–time.

The technique extends straightforwardly to multiple joins, by grouping each join to produce a hierarchical result. When there are no more joins on the root–node attributes, an ungroup operation is inserted into the strategy (analogous to a Sort that produces the desired ordering). For star–like subqueries that are executed with the central relation first, our cascade is more efficient than either a cascade of flat binary joins or multiway join (e.g., Ingres tuple substitution). The results were confirmed by analytical and experimental results.

As an additional benefit, multiple-join queries implemented with hierarchicalized execution plans are much less sensitive to selectivity estimates than non–hierarchicalized plans, since they avoid the multiplicative effect between individual joins.

Finally, we showed that multiway–join (in the style of Ingres) is not desirable as an extension to a classical query processor. A cascade of HJOIN seems superior in execution time, simplicity and implementation effort.

References


