Integrating Semi-Join-Reducers into State-of-the-Art Query Processors*

Konrad Stocker[1] Donald Kossmann[2] Reinhard Braumandl[1] Alfons Kemper[1]

[1]Universität Passau
D-94030 Passau, Germany
⟨*lastname*⟩@db.fmi.uni-passau.de

[2]Technische Universität München
D-81667 München, Germany
kossmann@in.tum.de

Abstract

Semi-join reducers were introduced in the late seventies as a means to reduce the communication costs of distributed database systems. Subsequent work in the eighties showed, however, that semi-join reducers are rarely beneficial for the distributed systems of that time. This work shows that semi-join reducers can indeed be beneficial in modern client-server or middleware systems – either to reduce communication costs or to better exploit all the resources of a system. Furthermore, we present and evaluate alternative ways to extend state-of-the-art (dynamic programming) query optimizers in order to generate good query plans with semi-join reducers. We present two variants, called *Access Root* and *Join Root*, which differ in their implementation complexity, their running times, and the quality of plans they produce. We present the results of performance experiments that compare both variants with a traditional query optimizer.

1 Introduction

In the last few years, a series of new query processing techniques have been developed in order to speed up the execution of queries in a distributed environment and/or to speed up the execution of decision support queries in a data warehouse. One of the techniques which have been rejuvenated are *semi-join reducers*. The basic idea is to apply *join predicates* early in a plan in order to reduce the size of intermediate query results and, thus, reduce the cost of other operations. In other words, the idea is to apply the same join predicates *twice* or more often in a query plan.

Semi-join reducers were originally proposed in the late seventies [BGW+81, AHY83] in order to reduce the communication costs of a distributed system. Obviously, semi-join reducers are only effective if the (redundant) semi-joins are cheap and result in a significant reduction of the size of intermediate query results. Studies in the early eighties showed that semi-join reducers were rarely attractive for the distributed systems and query workloads considered at that time. Recent work, however, showed that semi-join reducers (or similar techniques) are indeed attractive for specific kinds of queries, even in a centralized system. Variants of semi-join reducers are, for example, attractive to process certain kinds of star queries [SKB+98], top N queries [CK98], TID hash joins [MR94], or functional joins [BCK98]. Furthermore, we will show that semi-join reducers can be very attractive in modern client-server and middleware systems.

One of the fundamental difficulties is to optimize plans with semi-join reducers because the search space of possible plans with semi-joins is huge; much larger than the search space of plans without semi-joins. The typical approach is to take advantage of semi-join reducers by optimizing queries in several steps [YC84, YOL84, CY90]: (1) find an *optimal* semi-join reducer for each base table; (2) optimize the join order and methods for the (reduced) base tables. "Optimal" usually means to find semi-join reducers such that the base tables are filtered as much as possible. Obviously, such an approach will not produce good plans in most cases; it will over-eagerly generate plans with semi-join reducers and it will sometimes select the wrong semi-join reducers because the best choice depends on the join order and join

*This work was supported by the German Research Council (DFG) under contract Ke 401/7-1.

methods used. We, therefore, propose to extend a query optimizer and integrate semi-join reducer and join-ordering, etc. into a single query optimization step. We propose two ways to extend a state-of-the-art optimizer and thoroughly discuss the tradeoffs of these two ways.

The remainder of this paper is structured as follows. Section 2 gives examples and presents the results of performance experiments that demonstrate the usefulness of semi-join reducers. Section 2 also describes the search space of plans with semi-join reducers. In Section 3, the two ways to extend an existing state-of-the-art dynamic programming [SAC+79] based optimizer are described. Section 4 summarizes the results of performance experiments that compare these two approaches with state-of-the-art query optimization without semi-join reducers. Section 5 describes query optimizer variants that can be used for very complex queries. Section 6 discusses related work and Section 7 contains conclusions.

2 Motivating Examples and Search Space

In the following we present examples which demonstrate the profitable use of semi-joins in query evaluation plans. The examples consider current (distributed) database systems using replication and caching mechanisms. Though we concentrate on distributed client-server systems we show that semi-joins can be profitable in centralized database systems, too. To show query plans we use a standard query tree representation. Base tables are represented as leaves, and join operations are represented as internal nodes, both annotated with site descriptions.

2.1 Benchmark Environment

In order to demonstrate the significance of our examples, we carried out running time experiments with a 'real-world' distributed query engine [BKK99]. Cardinalities and sizes of the tables are presented in Figure 1. The cardinality of every intermediate result and of the final result is 10.000. No indexes are considered. Hash-joins are used with 500 KB buffer size. As a result, joins with Table A could be performed without partitioning, whereas partitioning is required in all other cases (grace-hash joins). We assume in the following that operators can be executed at client and server machines.

name	cardinality	size
A	1.000	0.1 MB
B – E	100.000	10 MB

Figure 1: Base Tables

2.2 Reducing Communication Costs in a Distributed System

First of all we address a more traditional application of semi-joins. The main reason for using semi-joins in distributed database systems is to reduce inter-site communication. Projected attributes are sent from one server to another. After performing semi-join reduction the reduced table is sent to the first server [AHY83, Bra84]. But all these approaches are beneficial for traditional distributed systems only in some cases [BG81].

Queries containing only functional joins (non-expansive joins[1]) produce only very small benefits or no benefits at all in traditional symmetric systems by applying inter-site semi-joins. Some particular examples with expansive joins are conceivable where the use of semi-joins is profitable.

Today, however, most distributed systems have a client-server architecture. We, therefore, concentrate on this architecture. Client-server systems differ from traditional (symmetric) distributed database systems by their restricted communication between sites. Clients can communicate with servers, whereas servers mutually cannot. Therefore running time benefits are much higher in client-server systems than in traditional systems when using additional joins or semi-joins. As an example consider the following

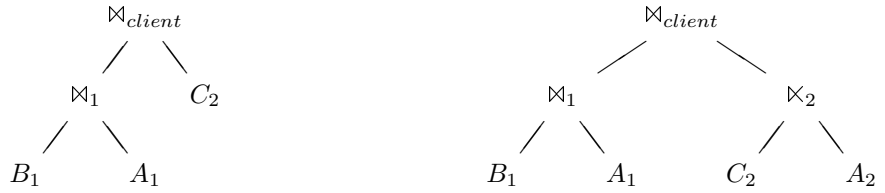
[1]Definition of 'expansive join': $|A \bowtie B| > \max(|A|, |B|)$.

SQL query and allocation schema:

```
SELECT *
FROM   A, B, C
WHERE  A.a = B.a ∧ A.b = C.b ;
```

Server - Site 1 : {A, B}
Server - Site 2 : {A, C}

Although *inter-site* semi-join techniques are not applicable in this case, a reduction in communication volume between Server 2 and the client by a factor of 10 can be achieved by using an *intra-site* semi-join. Figure 2 shows this situation. Base tables are indexed with the site they are read from, and operators ared indexed with the site they are executed at; e.g. $\bowtie_{client}$ indicates that the Join ist executed at the client. are The given running times below the plans show a speed-up of 5 using ISDN communication links between the client and the servers. Another set of experiments, with the client in Maryland (USA) and both servers in Passau (Germany) connected via the Internet, produced similar results (648 sec to 140 sec).



(a) traditional plan: 1632 sec (b) new plan: 322 sec

Figure 2: Client-Server System

2.3 Exploiting Machine Resources in a Distributed System

In this section we present an example for which additional joins lead to load balancing effects. In this example, communication costs are neglected because a high speed LAN is used. Figure 3 shows the use of additional joins to take advantage of (mostly unexploited) machine resources in distributed database systems. Consider the following SQL query:

```
SELECT *
FROM   A, B, C, D, E
WHERE  A.a = B.a ∧ A.b = C.b ∧
       A.c = D.c ∧ A.d = E.d ;
```

Server 1 : {A, B, C}
Server 2 : {A, D, E}

Using a traditional plan only one server executes joins, while the other server sends its tables to the client and stays idle. The client waits until tuples arrive from Server 1 and has to execute two expensive joins. In contrast, it is possible to exploit the computing resources of the second server also when an additional join using the replica of A is processed. The client only has to execute one join in this case. Figure 3 shows the results for two different configurations. In the first configuration, the client is as fast as the servers; in the second configuration, the client machine is significantly slower thean the server machines.

	Configuration A Server/Client: 86/86 MIPS	Configuration B Server/Client: 86/24 MIPS
traditional plan	161 sec	275 sec
new plan	101 sec	147 sec

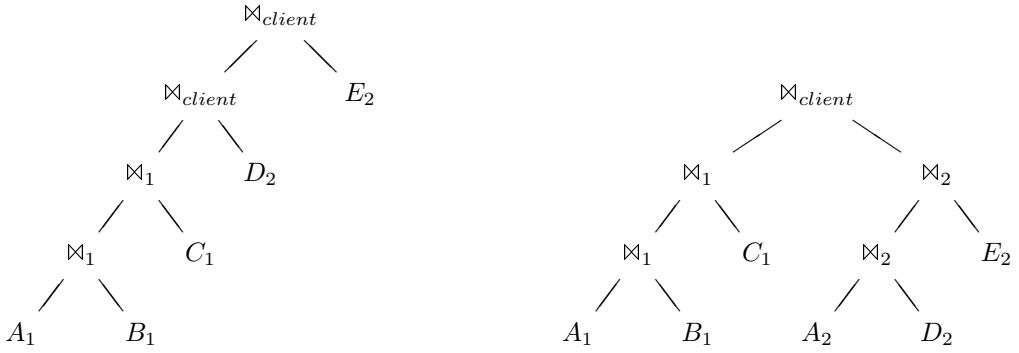
2.4 Reducing Disk IO in a Centralized Database System

Benefits due to semi-join reducers can also be achieved in centralized database systems. Consider the following example:

```
SELECT *
FROM   A, B, C, D, E
WHERE  A.b = B.a ∧ A.c = C.a ;
```

local Server : {A, B, C}
$|A| = 100, |B| = |C| = 100.000,$
$|A \bowtie B| = |C \ltimes A| = 10.000$

Figure 4 shows the measured results. Another case when additional semi-joins are profitable is their use as prerecordered filters to expedite subsequent join operations (e.g. $(A \ltimes B) \bowtie B$). A similar technique was studied in [Bra84].



traditional plan new plan

Figure 3: Exploiting Machine Resources

traditional plan: 69 sec new plan: 45 sec

Figure 4: Centralized System

2.5 Search Space of Plans with Additional Joins and Semi-Joins

The search space of a traditional query optimizer is determined by a number of different parameters. Join ordering, site selection and index selection have to be considered as well as choosing the most adequate join variants (nested-loop, sort-merge, hash). Generally we will extend the conventional search space in two directions: join-types used and join/semi-join orders.

2.5.1 Join and Semi-Join Operations

Consider the Tables A and B with attributes $attr(A)$ and $attr(B)$. The join predicates are denoted as p. We distinguish three cases.

1. **full-join** '$A \bowtie B$' (the standard join operation)
2. **semi-join** '$A \ltimes B$'. Only $attr(A)$ remains after the join. A semi-join can be considered as a generalized selection on A. Therefore $|A \ltimes B| \leq |A|$ holds.
3. **thin-join** '$A \bowtie^{thin_{b'}} B$' denotes a generalized way of joining tables where $attr(A)$ and a *subset* $b' \subset attr(B)$ is enclosed after the join. To preserve SQL duplicate semantics, b' must contain the key of B[2]. To reduce tuple-width all non-key and non-predicate attributes of B are projected.

In all cases, full-joins, semi-joins, and thin-joins can be implemented using any known join method (e.g. nested-loop, sort-merge, hashing, etc.).

2.5.2 Join and Semi-Join Orders

Generally, we consider the complete search space of all possible join and semi-join combinations. The use of semi-joins requires to allow multiple occurances of tables in query plans. Otherwise even simple

[2] $A \bowtie^{thin_{b'}} B$ can be denoted in relational algebra in the following way : $\Pi_{attr(A) \cup b'}(\sigma_p(A \times B))$ where $\emptyset \neq b' \subset attr(B)$. If b' does not contain the key of B idempotenz property could be lost: $A \bowtie^{thin_{b'}} B \neq (A \bowtie^{thin_{b'}} B) \bowtie^{thin_{b'}} B$.

semi-join plans like $(A \ltimes B) \bowtie B$ would not be possible. Plans like $A \bowtie B \bowtie B$ or $A \ltimes B \ltimes B$ have to be disregarded. Otherwise the search space grows infinitely large and enumerative algorithms do not terminate. But this termination problem can be solved by enumerating only *reasonable* plans, which is described in the following.

Avoiding Redundant and Unnecessary Joins and Semi-Joins Plans like $A \bowtie B \bowtie B$ or $A \ltimes B \ltimes B$ should not be considered, because no new predicates are applied. To allow only join operations at node n, which apply predicates not yet applied in a sub-tree of n, solves the problem. However, because of the anti-symmetric property of semi-join predicates ($A \ltimes B$ and $B \ltimes A$ must be distinguished) the predicate space is three times as big as using only full-joins[3].

It should be noted that this search space is much larger than the search space studied in previous work [BGW+81]. Previous work only considered semi-joins in order to reduce the size of base-tables before they are shipped to another site. For instance, the "new plan" of Figure 3 has not been considered in previous work.

3 Generating Query Plans with Additional Joins and Semi-Joins

3.1 Overview

In this chapter we present and discuss in detail the algorithms we studied. We will present algorithms which could be easily integrated into existing dynamic programming based optimizers. Our work differs from previous work since we do not focus on special query classes [BG81, YOL84] but on different search space and algorithm classes. For the purpose of presentation, we only consider select-project-join queries (SPJ queries). We will also concentrate on full-joins and semi-joins in the following and ignore thin-joins. Thin-joins, which are needed only in some particular cases (Figure 3), can be integrated in a similar way.

Basically we will present two different approaches. The first approach – we call it *Access Root* – can be easily integrated into existing optimizers. Plans are enumerated where semi-joins are used for reducing base-tables, not for reducing intermediate results. The second approach – we call it *Join Root* – has no restriction in applying semi-joins. Semi-joins can be applied at all query plan levels. This approach requires some changes in the traditional enumeration algorithm.

Obviously, there are many further approaches conceivable; e.g. post processing steps after traditional optimization, randomized algorithms, etc. Studying all these approaches is beyond the scope of this paper. In Section 5, however, we will present and evaluate some heuristics.

3.2 Classic Dynamic Programming

Before we present the *Access Root* and *Join Root* approach, we will describe the classic dynamic programming algorithm [SAC+79], which is used in most commercial state-of-the-art optimizers today. Figure 5 gives the dynamic programming algorithm which will be enhanced later on. The algorithm works in a bottom-up way as follows. First of all access-plans for all Tables R_i are generated (Lines 1 to 4). Such plans consist of operators like $table\_scan(R_i)$ or $index\_scan(R_i)$. They are inserted in a table-structure 'optPlan' which is set-indexed. This phase is called *access-root phase*. After that, in the following *join-root phase* (Lines 5 to 13) building-blocks of ascending size are produced. First 2-way joins by calling the joinPlans function on two access-plans, then 3-way join plans by combinations of all 2-way join plans and access-plans and so on up to n-way join plans.

[3] In some cases, plans like $A \ltimes (B \ltimes A)$ are profitable. For this reason the whole set of predicates P could be expressed as disjunct union $P = P_{equi} \dot\cup P_{left\_semi} \dot\cup P_{right\_semi}$. Let N be the set of nodes of a query plan. The term $w(n \rightarrow m)$ should denote a path in the query plan from n to m in data-flow direction (leave-node $\rightarrow$ root-node). The set of predicates applied at a node n is expressed by $p(n) \subseteq P$. The condition: $\forall n_1, n_2 \in N \wedge w(n_1 \rightarrow n_2) : p(n_1) \cap p(n_2) = \emptyset$ guarantees 'reasonable' joins and a finite search space. This condition must be modified when considering thin-joins because joining via key attributes is then often necessary more than once.

Input: SPJ query q on relations $R_1, \ldots, R_n$
Output: A query plan for q

```
1:  for i = 1 to n do {
2:      optPlan({R_i}) = accessPlans(R_i)
3:      prunePlans(optPlan({R_i}))
4:  }
5:  for i = 2 to n do
6:      for all S ⊆ {R_1, ..., R_n} such that |S| = i do {
7:          optPlan(S) = ∅
8:          for all O ⊂ S do {
9:              optPlan(S) = optPlan(S) ∪ joinPlans(optPlan(O), optPlan(S − O))
10:             prunePlans(optPlan(S))
11:         }
12:     }
13: return optPlan({R_1, ..., R_n})
```

Figure 5: (Classic) Dynamic Programming Algorithm

The advantage of dynamic programming in contrast to full enumeration is that it discards inferior building blocks after every step. This approach is called *pruning*. A (sub-) plan A is inferior to Plan B, if it is in relevant plan parameters at most as good but in at least one property worse than B. The relevant plan parameters are denoted as *plan properties*[4]. Only the best (comparable) plans are retained in optPlan, such that only these plans will be considered as building-blocks in later steps. If two plans are incomparable, both are retained in optPlan. For example, A *sort-merge-join* B and A *hash-join* B are incomparable if the *sort-merge-join* is more expensive than the *hash-join* because the *sort-merge-join* produces ordered results which might help to reduce the cost of later operations. Pruning should be carried out as early as possible to avoid the unnecessary enumeration of inferior plans. In the algorithm of Figure 5 all *bushy* plans are considered as an extension to the originally proposed left-deep variant by Selinger [SAC+79]; most commercial query optimizers that are based on dynamic programming do the same thing [GLSW94]. The complexity of this algorithm is $O(3^n)$ [OL90, VM96].

3.3 Access Root Variant

Figure 6 gives the modified variant of the dynamic programming algorithm shown above, where semi-joins are enumerated to reduce base tables. By adding lines N1 to N11 semi-joins are applied at the bottom-level of query plans immediately after the access-root phase. Semi-join plans are enumerated in a conservative way: Use a 'classic' dynamic programming optimizer to generate bushy plans as usual (Lines N1 to N8), but apply only semi-join operators instead of full-join operators (Line N5). Different semi-join variants like *nested-loop sj* or *hash sj* can be considered. The procedure called in line N9 traverses the entire dynamic programming table (optPlan structure) and moves all plans to their appropriate entries. For example, the plan $A \ltimes B$ enumerated in entry optPlan($\{A, B\}$) represents a subset of A and belongs therefore to entry optPlan($\{A\}$). Since many plans become comparable after moving, an additional pruning is performed (Lines N10 to N11). Reducing the number of plans in these base entries has an important effect on the algorithm's running time (see Section 4). Lines 5 to 13 remain almost unchanged. An adaption of cardinality estimation has to be done for correctness when considering semi-joins[5]. It must be mentioned that not all imaginable access-root-style plans are enumerated using this DP-based semi-join approach (see Section 3.3.2) .

3.3.1 Pruning Extension

While all plans contained in a standard DP-entry have the same result size this does *not* hold for plans containing semi-joins. That's why the pruning condition has to be modified to consider output cardinality too. Plan P can be pruned against Plan Q only if $|Q| < |P|$ in addition to all other pruning conditions

[4] examples: plan cost, relations contained, output attributes or sort order etc.

[5] For example, estimated_card($(A \ltimes B) \bowtie B$) = estimated_card($A \bowtie B$) holds.

```
Input:  SPJ query q on relations R_1, ..., R_n
Output: A query plan for q
1:    for i = 1 to n do {
2:        optPlan({R_i}) = accessPlans(R_i)
3:        prunePlans(optPlan({R_i}))
4:    }
N1:   for i = 2 to n do       // apply DP with semi-join operators
N2:       for all S ⊆ {R_1, ..., R_n} such that |S| = i do {
N3:           optPlan(S) = ∅
N4:           for all O ⊂ S do {
N5:               optPlan(S) = optPlan(S) ∪ SJoinPlans(optPlan(O), optPlan(S − O))
N6:               prunePlans(optPlan(S))
N7:           }
N8:       }
N9:   Traversal_and_MovePlans(optPlan)       // re-constitute optPlan - correctness
N10:  for i = 1 to n do
N11:      prunePlans(optPlan({R_i}))
5:    for i = 2 to n do
6:        for all S ⊆ {R_1, ..., R_n} such that |S| = i do {
7:            optPlan(S) = ∅
8:            for all O ⊂ S do {
9:                optPlan(S) = optPlan(S) ∪ joinPlans(optPlan(O), optPlan(S − O))
10:               prunePlans(optPlan(S))
11:           }
12:       }
13:   return optPlan({R_1, ..., R_n})
```

Figure 6: Access Root Algorithm

(e.g. cost(Q) $<$ cost(P)). For example, optPlan($\{A\}$) contains the Plan A and Plan $A \ltimes B$ which are incomparable because $|A| > |A \ltimes B|$ but cost(A) $<$ cost($A \ltimes B$). Hence both plans must be retained.

3.3.2 Search Space

Plans like $(A \ltimes B) \ltimes (C \ltimes B)$, which may be profitable, are not enumerated by the algorithm of Figure 6 because every table appears at most once in a reducing plan produced by dynamic programming. We experimented with a different *Access Root* variant which is more complex to implement and which does enumerate all possible semi-join reducers at the access-root level, and we saw that this variant has significantly higher running time and results in only slightly better plans.

3.4 Join-Root Variant

While the *Access Root* approach only considers a limited subset of possible and reasonable semi-join plans, the *Join Root* approach enumerates the complete search space of semi-join plans. Semi-joins are applied to base tables as well as to intermediate results at all query plan levels (e.g. $(A \bowtie B) \ltimes C$). Semi-join enumeration is fully integrated into the join-root phase, without a separate reduction phase. Therefore some crucial algorithmic modifications have to be done: enumeration of non-disjoint subsets, integration of a fix-point phase for plan completion and a consideration of anti-symmetric predicates to cover the complete search space. A new kind of pruning technique was also developed because ordinary pruning is not effective enough In the following we will discuss the algorithm presented in Figure 7 in detail by presenting major concepts.

3.4.1 Enumerator Extension

First, access-plans are generated as in classic dynamic programming (Figure 5). The term *table-set* denotes the set of tables involved in a plan. While traditional dynamic programming enumerates disjoint subsets of a table-set, the *Join Root* variant also enumerates all pairs of non-disjoint subsets of a table-set (Lines N1, N2, N3). Figure 8 gives an example for the entry optPlan($\{A, B, C\}$). On the left side disjoint

Input: SPJ query q on relations $R_1, \ldots, R_n$
Output: A query plan for q

```
1:   for i = 1 to n do {
2:       optPlan({R_i}) = accessPlans(R_i)
3:       prunePlans(optPlan({R_i}))
4:   }
5:   for i = 2 to n do
6:       for all S ⊆ {R_1, ..., R_n} such that |S| = i do {
7:           optPlan(S) = ∅
8:           for all O ⊂ S do {
N1:              for all P ⊂ O do {
N2:                  optPlan(S) = optPlan(S) ∪ joinPlans(optPlan(O), optPlan((S − O) ∪ P), 0)
N3:                  optPlan(S) = optPlan(S) ∪ SJjoinPlans(optPlan(O), optPlan((S − O) ∪ P), 0)
10:                 prunePlans(optPlan(S))
N4:              }
11:          }
N5:          timestamp = 0
N6:          do {
N7:              Δ = 'new plans with latest timestamp in S'
N8:              for all O ⊆ S do {
N9:                  optPlan(S) = optPlan(S) ∪ joinPlans(Δ, optPlan(O), timestamp+1)
N10:                 optPlan(S) = optPlan(S) ∪ SJjoinPlans(Δ, optPlan(O), timestamp+1)
N11:                 prunePlans(optPlan(S))
N12:             }
N13:             timestamp++
N14:         } while (Δ ≠ ∅)
12:      }
13:  return optPlan({R_1, ..., R_n})
```

Figure 7: Join Root Algorithm

pairs of sets are presented, which are generated by dynamic programming (symmetric expressions are not shown); on the right side the additional pairs of sets are shown, generated by the extended enumerator, which adds all subsets of a left subset to the corresponding right subset. Thereby, we are able to generate plans like $(A \ltimes B) \ltimes (C \ltimes B)$ as well as $(A \bowtie B) \ltimes C$, which cannot be enumerated by the *Access Root* variant presented before.

$\{AB\} \bowtie \{C\}, \{AB\} \ltimes \{C\}$	$\{AB\} \ltimes \{AC\}, \{AB\} \ltimes \{BC\}$
$\{AC\} \bowtie \{B\}, \{AC\} \ltimes \{B\}$	$\{AC\} \ltimes \{AB\}, \{AC\} \ltimes \{BC\}$
$\{BC\} \bowtie \{A\}, \{BC\} \ltimes \{A\}$	$\{BC\} \ltimes \{AB\}, \{BC\} \ltimes \{AC\}$
traditional DP	additional enumerated in extended DP

Figure 8: Traditional DP vs. Extended DP

3.4.2 Fix-point Iteration

In some cases an additional fix-point iteration is necessary to get *complete* (sub-) plans. A (sub-) plan is *complete* if it is result equivalent to a plan without semi-joins involving the same relations. Therefore a *fix-point iteration* for every optPlan entry is performed (Lines N5 to N14). Figure 9 gives an example of

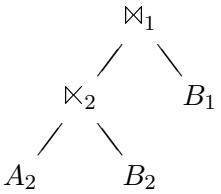


Figure 9: Fix-point Enumeration

a complete plan. The more semi-joins are applied in a left-deep building block, the higher the query plan becomes, the more iterations in the fix-point phase are needed for completion. Due to correctness con-

ditions (Section 2.5) the iteration terminates. To avoid re-enumeration of plans, enumerated in iterations before, time-stamps are added. Plans are initially marked with time-stamp "0" (Line N5).

3.4.3 Vertical Pruning

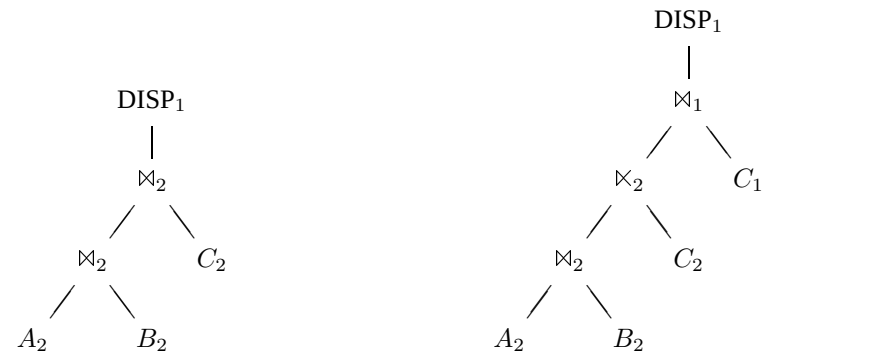
As well as in the *Access Root* algorithm additional plan properties must be considered for pruning. Since entries may contain plans with different output (e.g. $A \bowtie B$ and $A \ltimes B$), the output attributes of a plan must be considered; e.g. $A \bowtie B$ may not be pruned in favour of $A \ltimes B$ because $A \bowtie B$ produces a different output. This property is not considered for pruning in the classic dynamic programming algorithm, assuming that projections are pushed down.

One main problem of the *Join Root* variant is that comparable plans may be located in different entries of the dynamic programming table. For example consider the Plan $A \ltimes (B \ltimes D)$ in optPlan($\{A, B, D\}$). Comparable plans can be found in optPlan($\{A\}$), optPlan($\{A, B\}$), optPlan($\{A, C\}$), optPlan($\{A, D\}$) and optPlan($\{A, D\}$). Therefore, *intra*-entry pruning in the dynamic programming table is less significant, and we must also carry out *inter*-entry pruning. Thus, we have integrated an additional pruning step that we refer to as *vertical pruning*. The idea is to compare (and prune) plans between *all entries* which may contain comparable plans. We measured a running time speed-up of up to a factor of ten using vertical pruning.

3.4.4 Anti-symmetric Predicates

Dealing with semi-joins requires to distinguish different kinds of predicates. Semi-join predicates cannot be applied in a symmetric way like equi-join predicates, e.g. $A \ltimes B \neq B \ltimes A$. This anti-symmetric behavior has some effects on the applicability tests during enumeration (Section 2.5.2). Furthermore, in contrast to equi-joins, it may be advantageous to apply predicates several times in a query plan, e.g. $(A \ltimes B) \bowtie (B \ltimes A)$). Traditional dynamic programming produces only plans in which each predicate is applied only once.

3.4.5 Discussion



(a) traditional plan: 202 sec

(b) optimal plan (*Join Root*): 77 sec

Figure 10: Influence of Tuple-with

There are some cases, where only the *Join Root* variant is able to produce the best plan. Consider the following SQL query:

```
SELECT *
FROM   A, B, C
WHERE  A.a = B.a ∧
       A.b = C.b ∧
       B.c = C.c ;
```

Server 1 : {C}, Server 2: {A, B, C}
$|A| = |B|$ = 10K, tuple-size : 100 Byte
$|C|$ = 100, tuple-size : 1000 Byte
$\sigma_{AB} = \sigma_{AC} = \sigma_{BC} = 0.01$

The term σ_{XY} denotes the selectivity of the join predicate of $X \bowtie Y$. At first glance, it seems that the whole query should be processed at Site 2 without any semi-joins (Figure 10a). However, the transfer volume depends not only on the number of result tuples, but also on the *tuple-width*. Plan 10a transfers

the same number, but six times wider tuples than Plan 10b to the client. Plan 10b can only be produced by the Join Root algorithm. These examples show that the limited *Access Root* variant is often not sufficient for optimal plans, which can be found in the complete semi-join search space.

3.5 Selectivity and Cardinality Estimates

One difficult problem in query optimization is to find a correct or at least reasonable estimation of the selectivity of a join predicate. Traditional methods produce satisfying results only under certain circumstances (independence, uniformity, etc.) and fail when more predicates are applied simultaneously or in succession, because of correlations [vG93, SS94]. Even complex approaches do not produce satisfying results in these case [Yao77, KTY82, MCS88, SS94, GP89]. The more successive semi-joins are applied the higher the error in the estimating the result size becomes. That's why we take a different approach in our optimizer. We use the following conservative estimate:

$$|A \ltimes B| \approx min(|A|, |A \bowtie B|)$$

With this formula, an upper bound for the result cardinality is estimated. In doubt, no semi-joins will be used because the size of the semi-joins is overestimated. A more complex, statistic based computation model might produce plans with more semi-joins.

4 Experiments and Results

In this section we present the results of more performance experiments that we carried out. We measured algorithm running time and a large range of qualitative aspects of the different algorithms. Various benchmark parameters were changed to analyze the effects. Basically we studied the following parameters:

- join-graph topology (CHAIN, STAR, etc.)
- allocation schema (centralized, distributed, # sites, # replica)
- query-complexity (# relations, # predicates)
- cardinality distribution, tuple-width variance
- network topologies (symmetric, hierarchical)
- query processing resources (client speed, server speed, network bandwith, etc.)

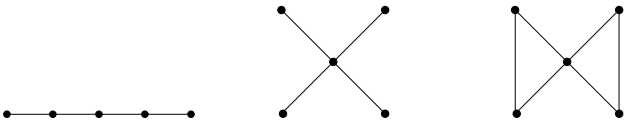


Figure 11: Join Graphs

The optimizer was executed on a Sun Ultra with a 167 MHz SPARC processor on SUN Solaris 2.6. We studied in general three query topologies shown in Figure 11. CHAIN and STAR topologies are well known in literature. The BONE topology is new and represents a STAR topology with additional predicates. For each topology we created 100 different settings for the cardinality of the base tables and the selectivity of the join-predicates. These settings were made at random following the approach proposed in [SMK97]. Using this approach, we were able to study a large range of different scenarios: queries with small and large base tables, low selectivity queries that produce many results, high selectivity queries that produce few results, and everything in between.

In the following we present the distilled results of our experiments. A ***benefit*** denotes a cost decrease of a plan by using additional semi-joins.

1. **Join Graph Topology:**
   This parameter has the most important influence on the magnitude of benefits. Different topologies produce very different benefits. The more predicates contained the more advantageous the *Join Root* approach becomes.

2. **Allocation Schema:**
   In centralized systems, only marginal benefits are achievable, substantial benefits can be achieved in distributed systems with a high degree of replication.

3. **Query Complexity:**
   The number of relations has a medium influence on the benefits. However, additional join-predicates increase the benefits by semi-joins enormously. The running time of the *Join-Root* algorithm increases dramatically with the number of tables involved in a query.

4. **Network topology:**
   Traditional symmetric communication paths lead to small benefits, whereas restricted client-server hierarchical architecture result in large benefits. The lower the bandwith the higher the benefits of using semi-join-reducers. Nevertheless, benefits can be achieved in any kind of network.

5. **Query-Processing:**
   Light-weight implementations of semi-joins are important. The *Join Root* algorithm takes more advantage from light-weight implementations than *Access Root* does. The slower the client (resp. the faster the servers) the higher the benefits.

We also measured the influence of cardinality distribution and high tuple-width variance in experiments. We saw that varying these parameters had no significant effect on scaled cost. Only in some cases the *Join Root* variant could take advantage from different tuple-widths. In the following we present running time and quality of plan results in more detail.

4.1 Running Time Experiments

Before comparing plan quality of semi-join algorithms with 'classic' dynamic programming we want to compare the running time of the proposed algorithms. Figure 12 gives an impression for the proportion of the different running times for a fixed parameter setting. The number of relations was varied on a STAR join topology in a client-server system with five servers and one client. It becomes clear that the different algorithms vary significantly in their running time. The low running time of the *Access Root* algorithm allows to apply this variant even on complex queries. On the other hand, when considering the complete semi-join search space, the application of the *Join Root* variant is limited to less-complex queries due to its high running time. Therefore we present some heuristic extensions for *Join Root* (also for *Access Root*) in Section 5 to process even complex queries.

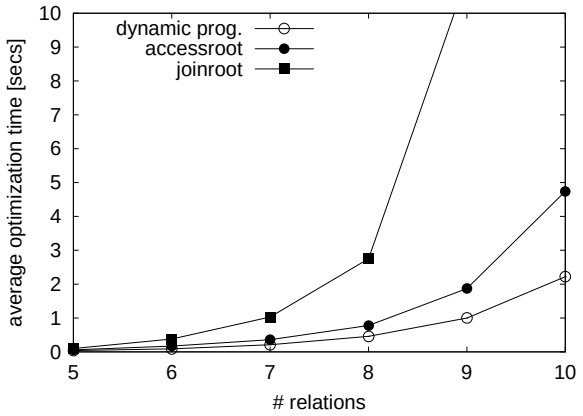


Figure 12: Algorithm Runtime

4.2 Quality of Plans

We measured the quality of the plans produced by the alternative algorithm variants for different topologies. To validate the quality of plans, we used the optimizer's cost model, which is an extension of the cost model used in [SMK97]. First we applied the *Join Root* variant to find the 100 optimal plans for each query and then we applied classic dynamic programming and the *Access Root* variant in order to find out how much, on an average, their plans were more expensive than the optimal plans. For example *average scaled cost (avgSC)* of 4.18 means that plans generated by classic dynamic programming are, on an average, four times more expensive than the optimal plan generated by *Join Root*. *Maximum scaled cost (maxSC)* of 42.9 shows that at least one plan is even 42.9 times more expensive than the optimal plan generated by *Join Root*. As default we studied a 5-way join query in a client-server environment with two servers (ISDN network). Varying other studied parameters like allocation schema, communication speed, etc., which are not shown in Figure 13, lead to comparable results.

	classic DP ($\neg$ SJ)		Access Root		Join Root	
	avgSC	(maxSC)	avgSC	(maxSC)	avgSC	(maxSC)
chain	1.65	(11.2)	1.0	(1.0)	1.0	(1.0)
star	4.18	(42.9)	1.0	(1.1)	1.0	(1.0)
bone	5.86	(107.6)	2.22	(12.5)	1.0	(1.0)

(a) join graph

	classic DP ($\neg$ SJ)		Access Root		Join Root	
	avgSC	(maxSC)	avgSC	(maxSC)	avgSC	(maxSC)
bone (symmetric)	1.64	(6.29)	1.37	(2.69)	1.0	(1.0)
bone (client-server)	5.86	(107.6)	2.22	(12.5)	1.0	(1.0)

(b) network topology

Figure 13: Algorithm comparison

In summary, we found that the use of additional of semi-join reducers is reasonable in all scenarios and environments. The use of *Access Root* or *Join Root* variant depends strongly on the join graph. *Access Root*, as a fast and easy implementation works well for many topologies and is feasible even for very complex queries, whereas *Join Root* achieves the best results in all topologies and is applicable up to medium query sizes. In the next section we show how the running time of the presented algorithms can be reduced for complex queries by making use of heuristics.

5 Dealing with Complex Queries

In this section, heuristics are presented and evaluated which reduce the running times of the *Access Root* and *Join Root* approach. Using these heuristics, *Join Root* may be applicable for more complex queries and the running time of the *Access Root* algorithm is almost as good as that of classic dynamic programming. It need not be mentioned that the use of heuristics can always impact the quality of plans, because optimal plans may not be considered by the heuristics.

Besides standard heuristics, which reduce the number of enumerated semi-join plans in a straightforward way (e.g. choosing the 'best' k semi-join plans), we also studied the application of Iterative Dynamic Programming Algorithm (IDP), a new class of algorithms to optimize very complex queries [KS00].

5.1 Heuristics

Obviously, the additional enumerated semi-joins lead to an increased running time of the algorithms. Hence we will try to find more restrictive criteria for the application of profitable semi-joins. In the following we present and evaluate conceivable approaches. We studied an 8-way STAR query in a client-server environment with two servers.

1. **Best-of Variants:** To reduce the number of enumerated plans, which depends basically on the number of all plans in an entry of the dynamic programming table, the number of plans involving semi-joins (semi-join plans for short) can be limited. Only the number of semi-join plans in base-table entries of the dynamic programming table must be limited in the *Access Root* algorithm, whereas all entries have to be considered in the *Join Root* algorithm. We studied two variants:

   (a) choose k *smallest* semi-join plans (least output tuples) and

   (b) choose k *cheapest* semi-join plans (least estimated cost)

   These variants were choosen because, in general, small intermediate results (variant (a)) and little cost overhead (variant b)) lead to the highest benefits. Figure 14 shows the results of variant (b). The term 'full' denotes no limitation on the number of semi-join plans ($k = \infty$).

star-8		classic DP ($\neg$ SJ)	$k = 1$	$k = 3$	$k = 10$	full
Access Root	avg. opt. time [secs]	0.4	1.0	2.0	2.1	2.2
	avg. scaled cost	9.6	5.2	1.2	1.0	1.0
Join Root	avg. opt. time [secs]	0.4	4.0	8.3	19.2	29.5
	avg. scaled cost	9.6	5.9	1.2	1.0	1.0

Figure 14: Best-Of Heuristic (Cost)

   It is interesting to note that keeping non-reduced plans is necessary. To consider only reduced plans for enumeration like proposed in previous work (fully reducers) leads, in general, not to the best query evalutation plan.

2. **Base Tables:** These heuristics only allow base-tables as semi-join partners. We denote B as semi-join partner in $A \ltimes B$. This heuristic produces acceptable plans (avg. scaled cost: 2.2) within a short running time (*Access Root*: 1.1 sec). We studied also other restrictions on semi-join partners like a limitation of their cost, but these restrictions do not produce good results.

3. **Iterations:** Figure 15 shows that limiting the number of iterations i during fix-point computation works very well. Even omitting the fix-point iteration altogether produces good results with a much smaller running time.

star-8		classic DP ($\neg$ SJ)	$i = 0$	$i = 1$	full
Join Root	avg. opt. time [secs]	0.4	15.03	26.5	29.5
	avg. scaled cost	9.6	1.0	1.0	1.0

Figure 15: Iteration Heuristic

To sum up, we saw that the iteration heuristics and the best-of heuristics produce very good results within a fraction of the full running time.

5.2 Iterative Dynamic Programming

Besides the presented standard heuristics we studied more sophisticated, adaptive approaches which produce as good plans as dynamic programming if dynamic programming is viable and as-good-as possible plans if dynamic programming turns out to be not viable. The main idea of `IDP` is to apply dynamic programming several times in the process of optimizing a query; either to optimize different parts of a plan separately or in different phases of the optimization process. `IDP` works essentially in the same way as dynamic programming with the only difference that `IDP` respects that the resources (e.g., main memory) of a machine are limited or that a user or application program might want to limit the time spent for query optimization. Due to a lack of space we refer the reader to [KS00] for a detailed description.

Figure 16 shows the running time/quality trade-off between alternative heuristics applied to the *Access Root* algorithm. Also an 8-way STAR query in a client-server environment with two servers was studied. Classic dynamic programming has a small running time but very high scaled cost (single filled

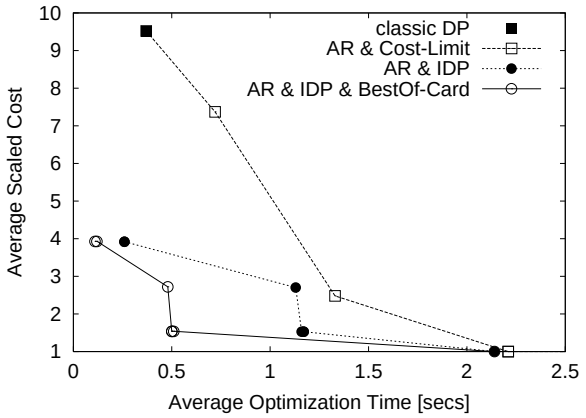


Figure 16: Heuristics *Access Root*

square). Whereas, limiting the cost of semi-join partners allows a more fine grained setting of the running time/quality trade-off (AR & Cost-Limit). Using Iterative Dynamic Programming instead of classic dynamic programming for the *Access Root* algorithm produces better results than most standard heuristics. But for the best performance both approaches, IDP and a good standard heuristic like Best-Of Card, should be combined (AR & IDP & BestOf-Card).

To summarize, the use of the proposed heuristics allows to apply both algorithms, *Access Root* and *Join Root*, even to complex queries despite the enormous increased search space. Using Iterative Dynamic Programming instead of classic dynamic programming enables to produce almost optimal plans involving semi-joins within the running time of classic dynamic programming without semi-joins.

6 Related Work

Semi-join reducers were originally proposed in the late seventies [BGW$^+$81, AHY83]. Replication was not mentioned in that work, and modern architectures like client-server or middleware systems were not considered at all. The work of that time focussed on finding optimal reduction schedules [HY79] – called *full reducers* [BG81], whereas local processing costs were neglected [JK84, HY79, AHY83]. Also the integration of proposed techniques into existing optimizers was not considered.

Most approaches proposed in the literature focussed on reduction of communication costs [HY79, AHY83]. In today's systems with high-speed networks, communication is often not the limiting factor. These approaches do not work well for modern architectures like middleware systems, where communication between servers is prohibited.

In addition, several approaches proposed at that time only work for certain classes of queries (e.g. tree queries) [BC81]. However, using different optimizers for different query classes is not acceptable for current database systems which need one optimizer for all queries. The algorithms proposed in this paper can be easily integrated into such an optimizer.

To date, most approaches that make use of semi-join reducers work in three phases [YC84]: (1) the copy identification phase, (2) the reduction phase, and (3) the assembly phase. These semi-join reducers are only applied in the second phase and they are only applied to base-tables [YOL84]. We avoid this strict separation in the *Join Root* algorithm.

Another problem addressed in this paper is the estimation of the size of intermediate results in the presence of semi-join reducers. Descriptions of existing approaches in this area can be found in [Ric81, KTY82, SS94, MCS88, PSC84, Chr84]. They are often very complex [GP89] and produce faulty results [vG93] on both ends; i.e., the estimates can be much too high and much too low. In contrast, we proposed a simple approach to estimate the size of the result of a semi-join (Section 3.5). Our approach can very easily be integrated into an existing optimizer, without changing the cardinality estimation routines for joins or other operators. Furthermore, our approach is *conservative*; i.e., it only errs on the high side.

7 Conclusion

In this paper we presented two new algorithms to integrate semi-join operations into existing state-of-the-art optimizers, which are mostly based on dynamic programming. First, we presented several examples, that demonstrate the usefulness of semi-join plans for today's distributed environments. We measured these examples in a real-world environment with a distributed query engine, using LANs, the Internet, and ISDN. Afterwards, two algorithms to generate plans with semi-joins – *Access Root* and *Join Root* – were presented. Necessary adaptions (and their effects) of a classic dynamic programming optimizer were described in detail. In particular, we showed how to estimate the cardinality of intermediate results in the presence of semi-joins. In addition, we carried out extensive performance experiments to compare the proposed algorithms with traditional query optimization.

Beside running time experiments, which gave an impression of additional cost by integration of semi-joins, different quality effects were analyzed. Generally, we measured high benefits by the use of semi-joins. We showed that the benefits depend on various parameters like query topology, network topology, network capacity, and query processing capabilities. In particular, we showed that real-world heterogeneous client-server or middleware systems yields the highest benefits.

Both algorithms work well with typical query profiles. However, when queries become very complex, the *Join Root* running time of exploiting the full search space becomes prohibitively high. To cope with this problem, we proposed different heuristic extensions for both algorithms. It became clear that running time and quality of plans can be adjusted in a fine granular way by choosing more or less restrictive heuristic parameters.

As future work, we plan to integrate *thin-joins* into our optimizer, whereby only moderately increasing the running time of query optimization. Furtheron, we are curious to see how this approach works with other query types; e.g. group by, top N, queries with expensive predicates, etc.

References

[AHY83] P. Apers, A. Hevner, and S. B. Yao. Optimization algorithms for distributed queries. *IEEE Trans. Software Eng.*, 9(1), 1983.

[BC81] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *Journal of the ACM*, 28(1):25–40, January 1981.

[BCK98] R. Braumandl, J. Claussen, and A. Kemper. Evaluating functional joins along nested reference sets in object-relational and object-oriented databases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 110–121, New York, USA, August 1998.

[BG81] P. A. Bernstein and N. Goodman. Power of natural semijoins. *SIAM Journal on Computing*, 10(4):751–771, November 1981.

[BGW+81] P. Bernstein, N. Goodman, E. Wong, C. Reeve, and J. Rothnie. Query processing in a system for distributed databases (SDD-1). *ACM Trans. on Database Systems*, 6(4):602–625, December 1981.

[BKK99] R. Braumandl, A. Kemper, and D. Kossmann. Database patchwork on the Internet (project demo description). In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 550–552, Philadelphia, PA, USA, June 1999.

[Bra84] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 323–333, Singapore, Singapore, 1984.

[Chr84] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Trans. on Database Systems*, 9(2):163–181, June 1984.

[CK98] M. Carey and D. Kossmann. Reducing the braking distance of an SQL query engine. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 158–169, New York, USA, August 1998.

[CY90] M.-S. Chen and P. S. Yu. Using join operations as reducers in distributed query processing. In *Proceedings of the Second International Symposium on Databases in Parallel and Distributed Systems*, pages 116–123, Dublin, Ireland, July 1990.

[GLSW94] P. Gassner, G. M. Lohman, K. B. Schiefer, and Y. Wang. Query optimization in the IBM DB2 family. Technical Report RJ9734, IBM Almaden Research Center, March 1994.

[GP89] D. Gardy and C. Puech. On the effect of join operations on relation sizes. *ACM Trans. on Database Systems*, 14(4):574–603, 1989.

[HY79] A. R. Hevner and S. B. Yao. Query processing in distributed database systems. *IEEE Trans. Software Eng.*, 5(3):177–187, 1979.

[JK84] M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

[KS00] D. Kossmann and K. Stocker. Iterative dynamic programming: A new class of query optimization algorithms. *ACM Trans. on Database Systems*, 25(1), March 2000. To appear.

[KTY82] L. Kerschberg, P. D. Ting, and S. B. Yao. Query optimization in a star computer network. *ACM Trans. on Database Systems*, 7(4):678–711, December 1982.

[MCS88] M. V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys*, 20(3):191–221, September 1988.

[MR94] R. Marek and E. Rahm. TID hash joins. In *International Conference on Information and Knowledge Management (CIKM)*, pages 42–49, Gaithersburg, Maryland, USA, 1994.

[OL90] K. Ono and G. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 314–325, Brisbane, Australia, August 1990.

[PSC84] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 256–276, Boston, USA, June 1984.

[Ric81] P. Richard. Evaluation of the size of a query expressed in relational algebra. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 155–163, Ann Arbor, USA, 1981.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–34, Boston, USA, May 1979.

[SKB+98] K. B. Schiefer, J. Kleewein, K. Brannon, G. M. Lohman, and G. Fuh. IBM's DB2 Universal Database demonstration. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, page 703, New York, USA, August 1998.

[SMK97] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208, August 1997.

[SS94] A. Swami and K. B. Schiefer. On the estimation of join result sizes. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, volume 779 of *Lecture Notes in Computer Science (LNCS)*, pages 287–300, Cambridge, United Kingdom, March 1994. Springer-Verlag.

[vG93] A. van Gelder. Multiple join size estimation by virtual domains. In *Proc. ACM SIGMOD/SIGACT Conf. on Princ. of Database Syst. (PODS)*, pages 180–189, Washington, D.C., May 1993.

[VM96] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian product. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 35–46, Montreal, Canada, June 1996.

[Yao77] S. Yao. Approximating block accesses in database organizations. *Communications of the ACM*, 20(4):260–261, April 1977.

[YC84] C. Yu and C. Chang. Distributed query processing. *ACM Computing Surveys*, 16(4):399–433, December 1984.

[YOL84] C. T. Yu, Z. M. Ozsoyoglu, and K. Lam. Optimization of distributed tree queries. *Journal of Computer and System Sciences*, 29(3):409–445, December 1984.