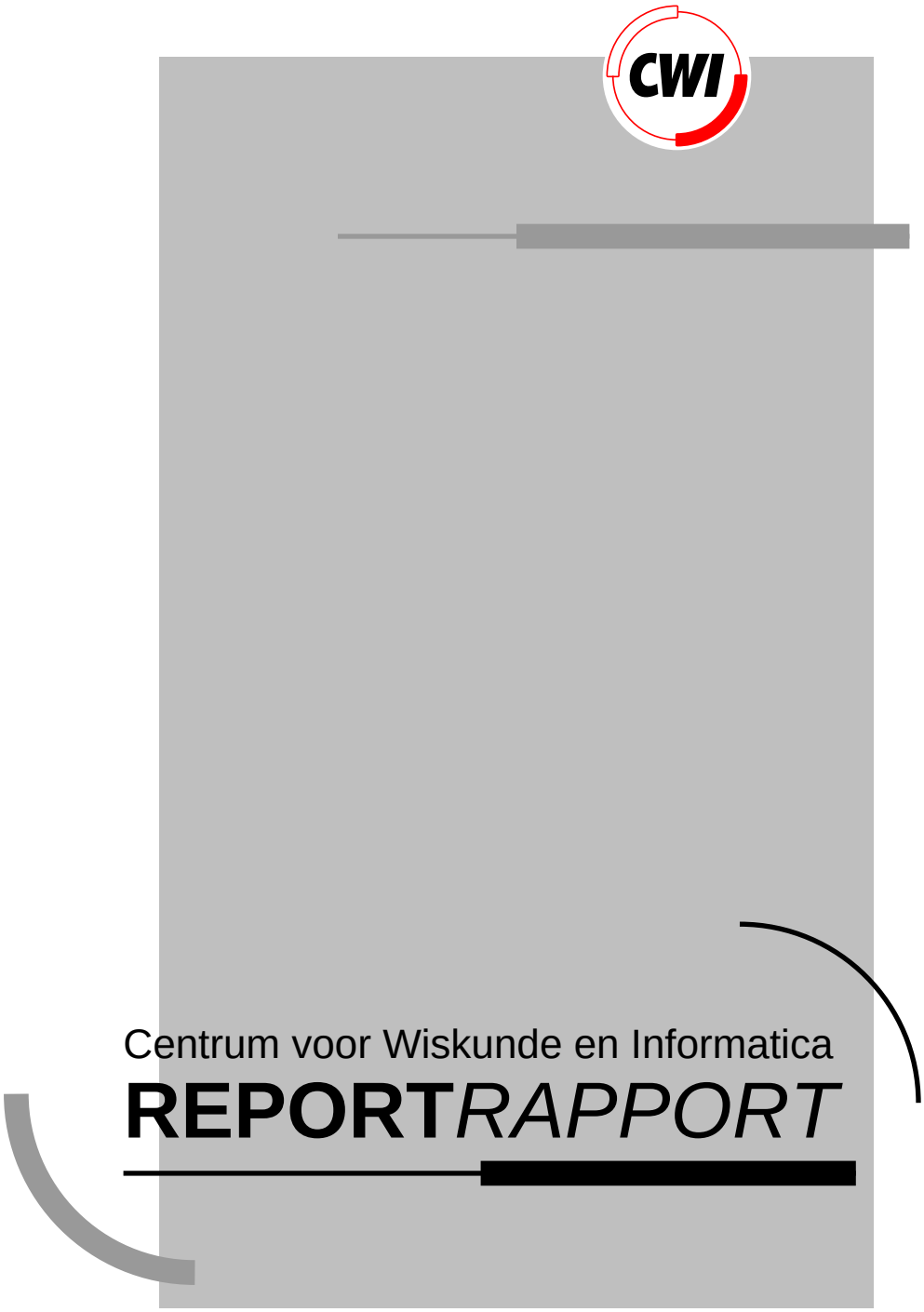

Centrum voor Wiskunde en Informatica

**REPORT***RAPPORT*

# Query optimization strategies for browsing sessions

M.L. Kersten, M.F.N de Boer

Computer Science/Department of Algorithmics and Architecture

**CS-R9268 1992**

# Query Optimization Strategies for Browsing Sessions

M. L. Kersten

CWI, P.O. Box 4079
1009 AB Amsterdam
The Netherlands

M.F.N. de Boer

CWI, P.O. Box 4079
1009 AB Amsterdam
The Netherlands

**Abstract**

*This paper describes techniques and experimental results to obtain response time improvement for a browsing session, i.e. a sequence of interrelated queries to locate a subset of interest. The optimization technique exploits symbolic analysis of the query interdependencies and retention of (partial) query answers. A prototype Browsing Session Optimizer (BSO) has been constructed that runs as a front-end to the Ingres relational system. Based on the experiments reported, we propose to extend (existing) DBMSs with a mechanism to keep and reuse small answers by default. Such investments quickly pay off in sessions with interrelated queries.* [1]

## 1 Introduction

In this paper we consider optimization strategies for the class of *browsing* sessions. A browsing session is a query sequence $Q_1 ... Q_n$ with common subexpressions. Overlap may occur on selection predicates that (partially) subsume queries issued later on in the session and repeating join expressions with different selection and projection conditions.

From an algorithmic and architectural point of view, browsing sessions are interesting because they address optimization and storage issues largely ignored in present relational systems. The state-of-the-art optimizers focus on optimization queries in isolation. Storage management and reuse of (partial) answers is left to the user, who explicitly manage their retention and the construction of access paths to improve response time of recurring and overlapping queries.

Few papers have been published on browsing sessions and their optimization. An early paper is [3][6], which describes a method for reusing answers to previous queries based on identifying common subexpressions. Our approach extends this work by also considering intermediate results produced during query evaluation for reuse and to provide experimental proof of their effectiveness.

A related research area is *multiple query optimization* (MQO), i.e. the simultaneous optimization of a set of queries. In MQO performance can be improved by identifying common subexpressions and to develop a plan for interleaved execution to maximize reuse of common results ([8],[12],[9],[11]). A prime difference with browsing is that MQO has complete knowledge on all queries in a session. In a browsing session this information becomes available incrementally, which makes re-ordering actions to amortize cost impossible and complicates resource management.

In this paper, we propose a common framework to capture the potentials for interquery optimization as materialized in browsing sessions. The framework is derived from the query graph representation in single query optimization environments. One graph describes the search space for optimization, another describes the actions taken by the browsing optimizer. Unlike the framework in [3] it supports partial subsumption relations. Moreover, it provides a basis for metrics to control resource management and to define a quantitative notion of optimization effectiveness later on. The strategies for browsing optimization are organized by the relational algebra operators. This leads to an orthogonal class of strategies, which can be analysed on their effectivity in isolation.

The main contribution of this paper is the experimental evidence obtained for performance improvement using a *browsing session optimizer* (BSO) that runs as a front-end of Ingres. It has been used to asses the performance gains of the strategies proposed and the exploitation of (temporary) results maintained automatically.

Several experiments have been conducted against

[1] This work is supported in part by the Pythagoras project (ESPRIT-III 7091)

the Wisconsin database to obtain quantitative indications for further research and development of BSO algorithms and DBMS kernel enhancements. It is is shown that this hybrid architecture already outperforms a conventional system when access paths are not available upfront, e.g. in large and broad tables, and for scientific browsing, e.g. location of an interesting subset for detailed analysis.

The paper is organized as follows. Section 2 introduces a model to characterise browsing sessions. Section 3 gives an orthogonal classification of optimization strategies. A preliminary performance model is presented in Section 4 and the architecture of our prototype BSO in Section 5. Section 6 presents the experimental results and Section 7 the attainable performance improvements with ad-hoc measures.

## 2 A model for browsing sessions

In this section we propose a model for representing browsing sessions. The main goal is to provide a canonical presentation of the opportunities for browsing optimizations and the decisions taken by a particular BSO.

### 2.1 Basic concepts

A browsing session is represented by a graph. The nodes correspond with queries issued by the user and directed arcs connecting query nodes to model possibly reuse of answers to construct part of a query answer. These arcs illustrate subsumption relations among selection predicates connected with join predicates. The graph is called a *query dependency graph* and satisfies the following properties:

**Definition 2.1** *Let $R$ and $S$ be two relations. Then $R$ is a partition of $S$, denoted by $R \preceq S$ if $R$ is a fragment of $S$ (by horizontal and vertical projection).*

**Definition 2.2** *Let $Q_0$ denote the database and $Q_i, Q_j$ $(0 < i < j)$ queries in a session then*

1. *$Q_j$ is strongly-related to $Q_i$, denoted by $Q_i \rightarrow_s Q_j$, iff $answer(Q_j) \preceq answer(Q_i)$*

2. *$Q_j$ is weakly-related to $Q_i$, denoted by $Q_i \rightarrow_w Q_j$, iff $\exists R \neq \emptyset : R \preceq answer(Q_i) \wedge R \preceq answer(Q_j)$.*

**Proposition 2.1**

*1)* $\forall i : Q_0 \rightarrow_s Q_i$
*2)* $\forall i : Q_i \rightarrow_s Q_i$ *(reflexivity)*
*3)* $Q_i \rightarrow_s Q_j \Longrightarrow Q_i \rightarrow_w Q_j$
*4)* $Q_i \rightarrow_s Q_j$ *and* $Q_j \rightarrow_s Q_k \Longrightarrow Q_i \rightarrow_s Q_k$
*5)* $Q_i \rightarrow_s Q_j$ *and* $Q_j \rightarrow_w Q_k \Longrightarrow Q_i \rightarrow_w Q_k$

The query dependency graph can be used to identify the nature of the session and opportunities for resource management. One extreme is the ideal situation where the user zooms in onto the relevant data, called a *convergent* session (See Figure 1). He selects a database fragment, which is subsequently reduced with more restrictive predicates. Each query $Q_i$ is strongly-related to its predecessors. A possible scenario for resource management here is to keep only the answers of the last two queries.

The worst case is a user searching for data without intentional overlap. This happens when the selection predicates are too selective, such as retrieving tuples by their key. In this case only a few accidental relationships exist and retaining answers for reuse is, in general, not effective. This is indicated as a *random* session.

The common case anticipated in scientific database environments is a coherent session, which indicates that queries may benefit from (partial) reuse of answers from several previous queries.

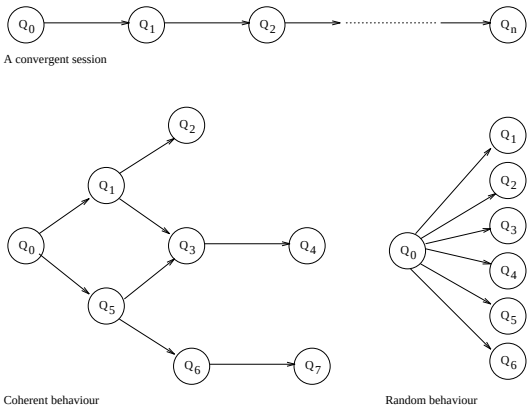


Figure 1: Query dependency graphs for different sessions

**Definition 2.3** *Let $Q_0$ denote the database and $Q_i, Q_j$ $(0 < i < j)$ denote the queries in a session. Then a query dependency graph is a directed graph $\mathcal{G}$=(V,E,Project,Select,R) with*
*$V$=$\{Q_0\} \cup \{Q_i\}$.*
*$E$=$\{(Q_i, Q_j) \mid Q_i, Q_j \in V$ and $Q_i \rightarrow_w Q_j\}$.*
*Select: $E \longrightarrow \{$selection predicates$\}$*

*Project:* $E \longrightarrow \{attribute\ lists\}$
*where* $Project((Q_i, Q_j)) = attr(Q_i) \cap attr(Q_j)$.
*R:* $E \longrightarrow \{weak, strong\}$
*where* $R((Q_i, Q_j)) = strong \Longleftrightarrow Q_i \rightarrow_s Q_j$
$R((Q_i, Q_j)) = weak \Longleftrightarrow \quad Q_i \rightarrow_w Q_j \wedge_i \rightarrow_s Q_j$

The edges are labeled with a selection predicate and projection list to capture the construction of the partially answer reused. That is, if $e$ is an edge from $Q_i$ to $Q_j$ then by definition $\pi_{Project(e)}\sigma_{Select(e)}(answer(Q_i))) \prec answer(Q_j)$. The graphical presentation of a query dependency graph ignores the properties that can be derived with Proposition 2.1. Moreover, the projection list is only shown if $attr(Q_i) \supset attr(Q_j)$ and the *strong* relationships are considered default.

### 2.2 The session graph

When a new query arrives the BSO must select an optimal evaluation plan and decide on the answers to be retained. The former task is supported by the query dependency graph. The new query is added to the graph and the BSO selects a subset of the relationships generated by this action for reuse. An algorithm to accomplish this task can be found in [6].

The BSO relies on heuristics to identify reuseable answers, because their future use is unknown. The heuristics take into consideration the query answer and useful intermediate results. The latter requires a large class of subqueries to be considered as well. Several strategies for this are introduced in Section 3 and the result leads to modifications of the query dependency graph.

After the final query dependency graph has been built, the browser selects an evaluation plan and decides on the answers to be retained. These decisions lead to a *session graph*, a subgraph of the query dependency graph, that administers replicated data fragments for response time improvement.

**Definition 2.4** *Let $Q_i, Q_j$ $(0 < i < j)$ denote the queries in a session. Then a session graph is a directed graph $\mathcal{G} = (V, E, Project, Select)$.*
$V = \{Q_0\} \cup \{Q_i\}$.
$E = \{(Q_i, Q_j) \mid Q_i, Q_j \in V \wedge Q_i reusedin Q_j\}$.
*Select:* $E \longrightarrow \{selection\ predicates\}$
*Project:* $E \longrightarrow \{attribute\ lists\}$
$Attr((Q_i, Q_j))$=$attr(Q_i) \cap attr(Q_j)$.

A useful metric for further analysis of a session graph is the *zoom rate*, which is the analogue for selectivity in query expressions. The zoom rate for a transition indicates the fraction of the source being reused. A small zoom rate indicates high return on investment when the target answer is being reused. It works under the assumption that in browsing sessions the probability of local access is high. Then the zoom factor can also be used as a low- and high-water mark for resource management.

**Definition 2.5** *Let $Q_i \rightarrow_s Q_j$ be an edge in the query dependency graph then the zoom rate is defined as:* $\Delta((Q_i, Q_j)) = \frac{size(answer(Q_j))}{size(answer(Q_i))}$

## 3 Browsing strategies

The goal of the browsing strategies is to identify useful results for future queries. The simpliest strategy to optimize a browsing session, called *naive*, is take the query formulation as is and to retain answers presented to the user only. That is, (intermediate) results of equivalent formulations are ignored. The effect is that the dependency graph is not exploited to its full extent. For example, consider the queries $Q_1$ and $Q_2$ below. Query $Q_2$ cannot be evaluated against $answer(Q_1)$, because $answer(Q_1)$ does not contain the *age* attribute any more.

| $Q_1$ | $Q_2$ |
|---|---|
| SELECT name | SELECT name |
| FROM PATIENTS | FROM PATIENTS |
| WHERE age $< 25$ | WHERE age $< 18$ |

To overcome these shortcomings, the browser optimizer could first decompose the query into a sequence of subqueries $Q_1, \ldots, Q_n$, such that $answer(Q) = answer(Q_n)$. These subqueries are evaluated and some answers are retained.

The challenge is to find a decomposition that maximizes response time reduction through reuse and whose subquery results are easy to manage. Unfortunately, the query components relevant for future steps are not a priori known in a browsing session. Therefore, we use heuristics based on the SPJ operators instead. Their combination covers a large class of browsing optimizer strategies.

### 3.1 Selection-based strategies

Zooming in onto a database subset is achieved by posing more-and-more restrictive selection predicates. A natural selection-based decomposition strategy is obtained by rewriting the selection predicate into a Conjunctive Normal Form. The lattice spanned by all subexpressions provides the search space for the

optimizer to replace the original query. Two complementary methods are considered.

The first method generates a subquery for each conjunct. Then the answer to the original query is the intersection of the subquery answers. The prime disadvantage is that many subqueries must be evaluated against the original table, which can be far more time consuming than evaluation of the query predicate. Moreover, additional costs are involved with the computation of the intersection to obtain the final result. Alternatively, the DBMS kernel can produce several intermediate results with a single scan or exploit parallel processing capabilities.
An alternative method is to cascade the selections onto each other in a pipe-lined fashion, i.e. query $Q_i$ is applied to the result of $Q_{i-1}$. This approach identifies a path through the lattice structure starting at a single conjunct and ending in the original predicate. The problem raised is to find the path with both minimal response time and improved reusability of intermediate results. A plausible heuristic is to order the conjuncts by decreasing selectivity. Then, at each step the browser can decide on the cost involved in retaining the answer as compared to recomputation.

Definition 3.1 gives a formal definition of the two selection based decomposition methods.

**Definition 3.1** *Let $Q$ be the query $\pi_A(\sigma_{p_1 \wedge \ldots \wedge p_n}(\bowtie (R_1, \ldots, R_m)))$, where each $p_i$ is a predicate without conjunctions. Then a selection based decomposition splits $Q$ into related queries $Q_1, ..., Q_n$, such that*

**method 1** *$Q_i$ is $\pi_A(\sigma_{p_i}(\bowtie (R_1, \ldots, R_m)))$, where $i \in \{1, \ldots, n\}$*
$answer(Q) = \bigcap_{i=1}^{n} answer(Q_i)$

**method 2** $Q_1 = \pi_{A_1}(\sigma_{p_1}(\bowtie (R_1, \ldots, R_m)))$
$Q_i = \pi_{A_i}(\sigma_{p_i}(answer(Q_{i-1})))$ *with* $i \in \{2, \ldots, n\}$
$A_i = A \cup \bigcup_{i \in \{1,\ldots,n-1\}} attr(p_{i+1})$
$A_n = A$

## 3.2 Projection-based strategies

Browsing sessions over scientific tables with hundreds of attributes mostly involve projections to limit the amount of data to be analysed. However, if the user changes his focus on new attributes within the same table partition then the answers to previous queries become useless.

To avoid re-computation of the selection predicate a browser optimizer can retain more attributes for reuse then strictly required. For example, it can retain all attributes in the temporary tables. Alternatively, it can retain the key of tuples selected to rejoin with the original relation to obtain missing attributes [6].

**Definition 3.2** *Let $Q$ be a query of the form, $\pi_A(\sigma(\bowtie (R_1, \ldots, R_n)))$, where $A$ is a set of attributes. Then projection based decomposition splits $Q$ into two strongly related subqueries $Q_1$ and $Q_2$, such that*
$Q_1$ *is* $\pi_{A'}(\sigma(\bowtie (R_1, \ldots, R_n)))$,
*where* $attr(\bowtie (R_1, \ldots, R_n) \supset A' \supset A$
$answer(Q_2) = \pi_A(answer(Q_1))$

Reconsider queries, $Q_1$ and $Q_2$. They can be split according to this strategy in the following strongly related queries. Now $answer(Q_{1,1})$ can be reused to directly evaluate $Q_{2,1}$. Answer($Q_{2,1}$) is more compact and a join with PATIENTS on the patientnr provides access to the remaining attributes. Figure 2 shows the corresponding session graph.

| $Q_{1,1}$ | $Q_{2,1}$ |
|---|---|
| SELECT * INTO TMP1<br>FROM PATIENTS<br>WHERE age $<$ 25 | SELECT patientnr,name<br>INTO TMP2<br>FROM TMP1<br>WHERE age $<$ 18 |
| $Q_{1,2}$ | $Q_{2,2}$ |
| SELECT name<br>FROM TMP1 | SELECT name<br>FROM TMP2 |

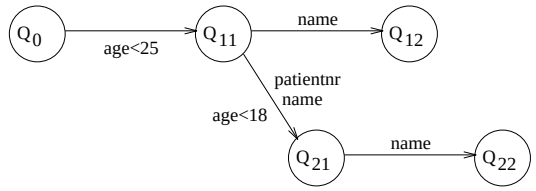


Figure 2: Projection based decomposition

## 3.3 Join-based strategies

Conventional query optimizers push selections and projections through the join operator to improve response time. However, in a browsing session this heuristic may not improve remaining queries. Instead, the joins exploit mostly functional dependencies to reconstruct meaningful information entities. Then, it may be more appropriate to perform the join first (or to build a join index) and to perform the selection afterwards. Subsequent queries involving the same functional dependencies, but different projections and selections, can reuse the join result without recomputation.

For example, a medical researcher interested in a correlation between a specific disease and the medical history of patients, could first join the relations PATIENTS and MEDICAL_HISTORY. Thereafter, browsing is limited to a single table against which we can apply the previous decomposition heuristics.

Given the complexity to evaluate a join, it is a challenge to the DBMS kernel designers to keep and reuse the intermediates produced in the process. For example, hash tables and tuple identifier lists constructed in the process should not be discarded immediately, but kept around for reuse.

**Definition 3.3** *Let $Q$ be a query of the form $\pi(\sigma(\bowtie(R_1, \ldots, R_n)))$ .*

*Join based decomposition splits $Q$ into $n$ subqueries such that*

$Q_1 = R_1 \bowtie R_2$.
$Q_2 = answer(Q_1) \bowtie R_3$.
$\vdots$
$Q_{n-1} = answer(Q_{n-2}) \bowtie R_n$.
$Q_n = \pi(\sigma(answer(Q_{n-1})))$.

This join-decomposition scheme is analoguous to the selection decomposition scheme, they both cascade individual operators. Moreover, the definition given here is an example of a much larger class of possible join decompositions, such as bushy, left-linear, etc.. Alternatively, the join- and selection-based strategies can be combined, such that $Q_1$ performs the join only. Thereafter a naive or selection strategy can be applied.

After the decomposition, the BSO decides on the results to be retained. This decision can be based on the metrics indicated in Section 2.2 and general resource management policies, such as FIFO, LRU, and reconstruction cost. A detailed analysis is beyond the scope of this paper.

## 4 Performance estimation

In the context of single query optimization the mechanisms to predict the performance has been thoroughly studied. A general performance model for a browser optimizer is beyond the scope of this paper and the state of the art. However, for the subclass of convergent browsing sessions we can formulate an analytical model to differentiate our BSO from a conventional optimizer.

Consider a convergent browsing session $Q_i$ of length $i (i > 0)$ with constant zoom factor $\Delta$ and let $N$ denote the number of pages in the table. Moreover, the table is not supported by indices. Since in both techniques the same information is sent to the user and both involve similar overhead in query startup, we focus on the pages read to locate the subset of interest and the page accesses for managing query answers.

In a conventional approach each query $Q_i$ requires a table scan, which costs $N$ pages accesses and the total session cost becomes $iN$.

The BSO involves a more extensive query analysis phase to identify reusable answers and the creation of a temporary table to hold it. This overhead of dictionary management is considered constant ($< 55$ms).

Due to implementation restrictions, our browser optimizer also stores the query answer before it is being sent to the user. Although this cost factor can be reduced considerably in an enhanced DBMS kernel, it is included in the model. Thus, the cost of a query $Q$ is the sum of four components:

Cost of scanning the relation: $R(Q)$.
Cost of storing $answer(Q)$: $A(Q)$.
Cost of scanning $answer(Q)$: $A(Q)$.
Cost of browsing overhead: $\gamma$.

$$cost(Q) = R(Q) + 2A(Q) + \gamma \tag{1}$$

Since each query reduces the relation with a factor $\Delta$ we have the following equality:

$$\forall i : A(Q_i) = \Delta R(Q_i) \tag{2}$$

Each query can be executed against the answer of the previous query, which leads to

$$\forall i > 1 : R(Q_i) = A(Q_{i-1}) \tag{3}$$

To make this equality apply to the case $i = 1$, we define

$$A(Q_0) = N \tag{4}$$

Now, the following can easily be proven by induction.

$$\forall i : R(Q_i) = \Delta^{i-1} N \tag{5}$$

Take $i = 1$, then

$$R(Q_1) \stackrel{(3)}{=} A(Q_0) \stackrel{(4)}{=} N$$

By induction we have for $i = n > 1$

$$R(Q_n) \stackrel{(3)}{=} A(Q_{n-1}) \stackrel{(2)}{=} \Delta R(Q_{n-1}) \stackrel{!}{=} \Delta^{n-1} N$$

From (2) and (5) follows

$$\forall i : A(Q_i) = \Delta^i N \tag{6}$$

Combining (1), (5) and (6) gives

$$\begin{aligned} \forall i : cost(Q_i) &= \Delta^{i-1}N + 2\Delta^i N + \gamma \\ &= \Delta^{i-1}N(1+2\Delta) + \gamma \end{aligned}$$

Straightforward summation leads to a cost for the session of length $n$ with zoom factor $\Delta$

$$\begin{aligned} \sum_{i=1}^{n} Q_i &= \sum_{i=1}^{n}(\Delta^{i-1}N(1+2\Delta)+\gamma) \\ &= n\gamma + N(1+2\Delta)\frac{\Delta^n - 1}{\Delta - 1} \end{aligned}$$

The curve for *cost* is exponentially decreasing and the BSO wins if

$$nN > n\gamma + N(1+2\Delta)\frac{\Delta^n - 1}{\Delta - 1} \qquad (7)$$

This situation depends on the session length and zoom factor.

## 5 Prototype SQL browser optimizer

To evaluate the relative performance of the strategies, we have implemented a browser session optimizer for a relational DBMS. A front-end implementation has been chosen, because we can not change the optimizer/kernel of the commercial version of Ingres. Moreover, it provides a mechanism to assess other relational systems and independent query optimizers [10] later on. Currently, the Ingres is used as our back-end DBMS.

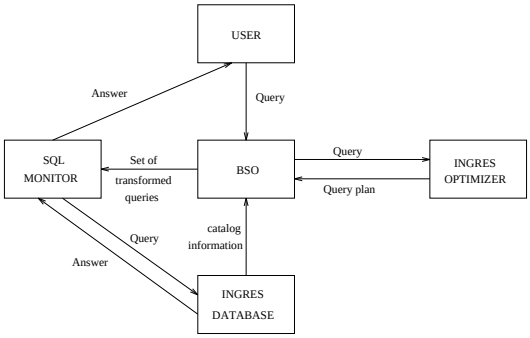


Figure 3: Browser optimizer architecture

The architecture is shown in Figure 3 where a box represents a process and arrows represent data flow. The browser accepts a flat SQL query without aggregates and transforms it into an equivalent batch of SQL requests that construct and exploit the (partial) answers. This transformation consists of two steps. First, a selection is made among the (partial) answers to solve the query with the best response time. Second, the best query plan is taken into execution and the browsing optimization strategy determines what (intermediate) results are retained.

Two alternatives exist to retaining (partial) answers. The browser can keep the answers in main memory or store them in the database. The former behaves as a private main-memory relational cache with fast response times. The disadvantage, however, is that the browser must be a relational DBMS itself, which greatly exceeds the manpower available for this project. Therefore, we have opted for the DBMS storage alternative. The browser keeps track of additional information, such as the query text, its (partial) answer(s) size, and the query dependency graph(s).

The browser optimizer currently uses a brute force approach to find the best execution plan. Therefore, it generates the set of equivalent plans by replacement of portions with the answers retained from previous queries. Their evaluation cost is determined using the DBMS query optimizer. For Ingres we merely send the query to the server and retrieve the evaluation plan considered best by its optimizer. From this plan the CPU and IO costs can be easily extracted.

## 6 Experimental results

The prototype browser optimizer has been used to experiment with the naive and decomposition-based strategies for convergent sessions against the Wisconsin benchmark database [5]. This approach provides valuable hints for extension of optimizers in state-of-the-art relational systems. Moreover, it provides a basis to compare browsing optimization with more traditional techniques. In particular, a comparison with ad-hoc browsing support, i.e. building an index first, is described in Section 7.1.

The client/server architecture of the DBMS made us use its built-in performance function *dbmsinfo()* to obtain the (cumulative) CPU, IO and elapsed time. Since the browser and the conventional system deliver the same output to the user, we have redirected it to /dev/null and, furthermore, ignored the buffered IO requests for communication of the server with the client process.

The performance figures illustrate both a per query and cumulative response time. They represent average values over several runs ignoring the largest and lowest value encountered.

### 6.1 Naive-based browsing

The base line for a browsing optimizer is to improve performance for a convergent browsing session. That is, each query can efficiently be evaluated against a (cached) old result. A session that captures this behavior is given by:

$Q_i$ = `SELECT * FROM table`
WHERE unique1¡cardinality(table)* $\Delta^i$

where $i$ denotes the i-th step in the session ($i < 8$) and $\Delta$ a constant zoom rate (0.0 .. 1.0). The table storage structure is a heap, which reflects the situation in many scientific tables where it is not feasible to maintain indices on all attributes. This storage structure causes the system to scan the table for relevant tuples.

The results, shown in Figure 4, are prototypical for the browsing sessions being analysed. That is, the initial investment is high, because the user has to wait for the answer of $Q_1$ to be stored, but soon the response time drops below the no-strategy line. In particular, a small zoom factor warrants investments for browsing optimization. However, for $\Delta = 0.5$ the investments are not recovered in a session over 8 queries.

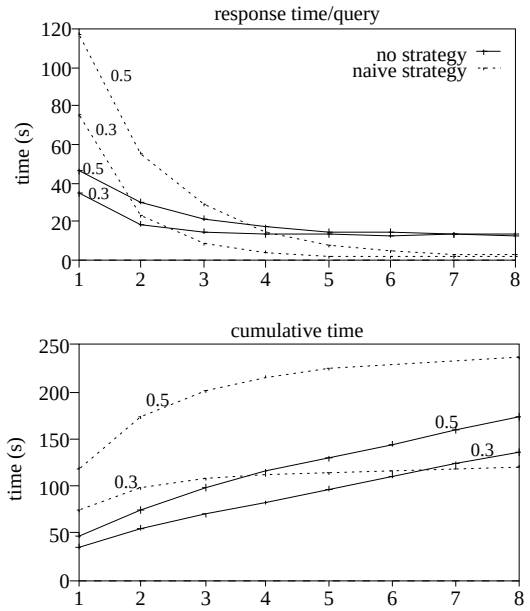


Figure 4: Naive-based browsing session

The elapsed time for queries $Q_5$-$Q_8$ of the naive strategy with $\Delta = 0.3$ is nearly constant, retaining these answers does not improve performance. The time observed reflects the minimum to handle the request and it implies that a browser optimizer should ignore results whose computational cost is less than a tunable system constant.

### 6.2 Browsing with projection

The response time of the previous experiment strongly depends on the number of attributes being retrieved and we expect a user to restrict the search to the attributes of interest in query $Q_1$. Then, the cost to retain the answers drops significantly and an even better response time for $Q_2$-$Q_8$ results. For example, consider the performance of the session described by:

$Q_i$ = `SELECT unique1, unique2`
FROM table
WHERE unique1<card(table)* $\Delta^i$

The zoom factor $\Delta$ is not constant during the entire session. The first query zooms much faster, because it combines a *horizontal* and *vertical* partitioning. Thus, query $Q_1$ produces $0.04\Delta$ of the original size and the overhead for storing it becomes small. The remaining queries zoom with a constant rate $\Delta$.

The results (Figure 5) illustrate again the high cost of producing the initial temporary. However, the naive strategy leads to a large cumulative performance improvement. The session is 68% faster than a traditional approach. A projection-based strategy with $A' = *$, i.e. first the vertical relation slice is being produced, is less effective. It leads to a 17% improvement over the traditional case.

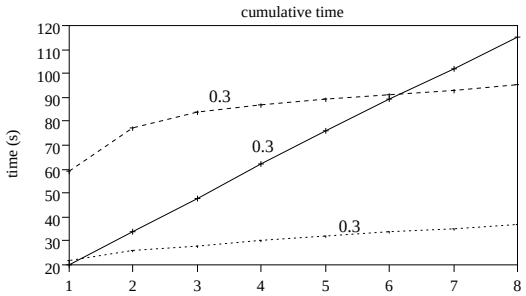


Figure 5: Session with projections

Furthermore, query $Q_2$ is already processed 3x faster under the naive strategy. Similar results for other $\Delta$s warrant automatic storage of a small results for potential reuse by a DBMS. The overhead is small (in this experiment only 10%) and a performance gain results when reused once.

The shortcoming of the naive strategy is that it only retains attributes in the projection list. Then,

selections over non-retrieved attributes cause the other queries to be evaluated against the original table. The projection-based strategy overcomes this disadvantage by identifying the attribute subset to be retained in the answer. Without circumstantial information it will retain all attributes used in the query. For example, the results for the session

$Q_i$ = `SELECT unique2 FROM table`
WHERE unique1<card(table)* $\Delta^i$

illustrate a performance degradation for the naive strategy of 15%, while the projection based strategy improves performance (14%).

Both experiments illustrate that the browser should not simply keep all attributes in an intermediate structure. Additional knowledge is required to predict attribute grouping in a result.

### 6.3 Join-based browsing

Queries mostly involves a join over several relations and a conventional DBMS processes the selections and projections on the operands before the join operation itself. To test this heuristic in a browsing environment, we have experimented with sessions where the predicate is evaluated before and (necessarily) after the join.

The first join experiment simulates exploitation of a functional dependency by equi-joining two Wisconsin tables on their *unique1* attributes. Moreover, we perform a convergent browsing with zoom factor 0.5 and 0.3.

$Q_i$ = `SELECT *`
`FROM table1, table2`
`WHERE table1.unique1 = table2.unique1`
`AND table1.unique3` $< card(table) * \Delta^i$
`AND table2.unique3` $< card(table) * \Delta^i$)

This session can be processed with a join-based, selection-based, and naive optimization strategy. The join-based strategy first computes the join, thereafter the queries can be solved by scanning the answer. The same effect is obtained with a naive strategy for query $Q_2$-$Q_8$, which are processed against the answer of $Q_1$. However, query $Q_1$ performs slightly better under the naive strategy, because the selection is processed before the join operation itself.

The performance results (Figure 6) illustrates the decision of the browser optimizer to ignore the intermediate results after $Q_5$. It re-uses the join result for query $Q_2$-$Q_4$; queries $Q_5$-$Q_8$ have been computed directly against the table without keeping the result.

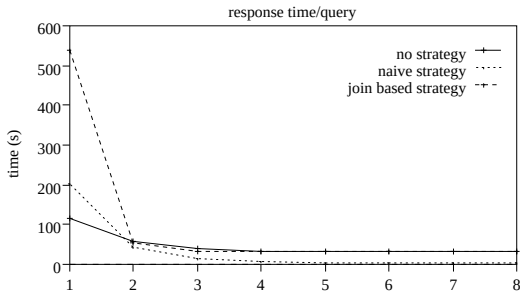


Figure 6: Join-based browsing session

## 7 Ad-hoc browsing support and further improvements

The browsing optimization techniques and experiments are focussed on large flat tables. This assumption generally does not hold for (scientific) databases. Rather, some indices already exists or one might be tempted to recluster the table or to create an index to speed-up the browsing session. In this section we show that such ad-hoc support is not necessarily better than the browsing optimizer proposed.

### 7.1 Comparison with ad-hoc indexing

If a large table is to be used in a browsing session, the user is generally adviced to recluster the data or to build an index to improve subsequent retrieval performance. However, index construction is an expensive operation, whose investments become profitable only after many selections.

To assess the performance impact of such ad-hoc support, we have compared our browsing strategy against a clustered and non-clustered index for the first experiment.

Modification of the Wisconsin table into ISAM on the search experiment is time consuming. It takes 187 seconds on our experimental platform, a SiliconGraphics multiprocessor machine with 6 CPUs and 64 Mb of memory. Once this index is available, the experiment ($\Delta$=0.3) can be processed within 10 seconds. However, our browser optimizer leads to a cumulative response time of 120 seconds, which is still about 40% faster.

Construction of a non-clustered index is much cheaper in the DBMS. It takes $\approx$ 45 seconds, thereafter, queries $Q_2$-$Q_8$ are processed within a few seconds. A rerun of experiment 1 and 2 show in this situation a looser and a winner for the BSO, respec-

tively. In particular, the second experiment benefits by trimming the answer to the attributes of interest only. Therefore, further work is required to support ad-hoc indexing using the techniques presented.

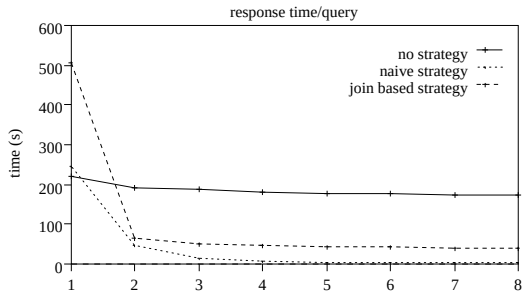


Figure 7: Predefined Join index ($\Delta = 0.3$)

### 7.2 Indexing support for joins

Index construction and exploitation also play a crucial role in processing join operations efficiently. In particular, our hypothesis that in browsing sessions most joins involve functional dependencies threatens the effectiveness of a BSO. For, the target attribute of the dependency (the referenced key) is often already supported by an index.

To assess the impact on the browser Figure 7 shows the effect of the first join experiment ($\Delta$= 0.3) with an index on the foreign key. In both cases the BSO outperforms the conventional approach. The high initial investment required for the join-based decomposition stems from the large intermediate size. The response time of $Q_2$-$Q_8$ is nearly constant, because the join results produced in $Q_1$ is repeatedly scanned. The improvements obtained with the naive strategy are caused by subsequent reduction of the intermediates.

### 7.3 Analysis of the cost components

The response time for the BSO experiments combine several cost factors, such as the cost to scan the relation(s), the cost to store the answer, the cost to scan the answer, and the cost to send the answer to the user. Their individual contribution to the response time is shown in Figure 8 as a breakdown for the different cost components for the first experiment with $\Delta$=0.3. The overhead caused by the BSO implementation on a black-box DBMS is shaded.

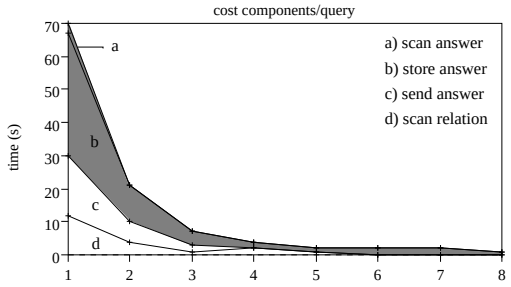


Figure 8: Response time component cost (naive strategy, $\Delta = 0.3$)

The high overhead for the BSO is indicative for possible significant improvements by re-engineering parts of the DBMS optimizer. For example, the BSO implementation can not advice the DBMS to handle the temporaries produced as scratch relations. They are indistinguishable from assignments to a permanent relation. However, a DBMS kernel geared towards browsing support can concurrently sent the result pages to the user and to the disk in a reusable format. This modification avoids a scan of the temporary relation and avoids most of its construction cost.

Based on the common architecture of relational DBMS kernels and the experimental results obtained, we are convinced that browsing sessions can be supported with minor changes to the DBMS kernel. In particular, it should take an attitude to retain temporary and intermediate results until they become outdated or when storage resources are exhausted.

## 8 Conclusions

This paper explores the nature of browsing sessions over a relational database and provides a framework to classify optimization strategies. Its main contribution, compared with previous results in this area, is the body of performance results obtained with a prototype browsing session optimizer running on top of a commercial DBMS.

The experiments show that simple optimization techniques, based on the credo not to discard (intermediate) results quickly, lead to significant improvements for total response time. In particular, small results should be kept around for reuse by default. Moreover, if the convergence of the session is a priori known then it pays off to invest resources in the first query to reduce response time of remaining queries.

The experiments also indicate that a browsing optimizer component can outperform ad-hoc optimization techniques, such as building secondary indices. Moving these decisions to the optimizer simplifies data management for both the casual user in his search through a large (scientific) database and the DBA to balance requests for permanent access paths.

Although this study has been largely focussed on convergent browsing sessions, the potential gains can be obtained with straight forward extensions to conventional query optimizers. They require two new cost factors, namely, the zoom rate ($\Delta$) and the fixed overhead caused by to managing the partially replicated database. Furthermore, the algorithms to choose a query evaluation plan should extended to use the replicated fragments.

Future research could focus on the resource management issue introduced by a browsing optimizer and dynamic partial re-clustering as an alternative to maintain the relation replicas. Our next step is to extend this study to improve a query optimizer for a parallel DBMS [10].

## References


[1] I. M. Author, "Some Related Article I Wrote," *Some Fine Journal*, Vol. 17, pp. 1-100, 1987.

[2] A. N. Expert, *A Book He Wrote,* His Publisher, 1989.

[3] J.R. Alsabbagh, V.V. Raghavan : "A Framework for Multiple-Query Optimization" *Proceedings of the workshop on Research Issues on Data Engineering: Transactions and Query Processing*, Feb 1992, pp. 157-162.

[4] S. Ceri, G. Gottlob, G. Wiederhold: "Interfacing Relational Databases and Prolog Efficiently",*Proceedings of the first international Conference on Expert DBS's*, April 1986.

[5] D.J. DeWitt, "The Wisconsin Benchmark", in *The Benchmark Handbook for database and transaction processing systems* (by J. Gray), Morgan Kaufmann Publishers Inc., 1991.

[6] S. Finkelstein: "Common Expression Analysis in Database Applications", *ACM SIGMOD Record*, 1982,p235-245.

[7] J. Gray: "The Benchmark Handbook for database and transaction processing systems", *Morgan Kaufmann Publishers Inc.* , 1991.

[8] M. Jarke: "Common Subexpression Isolation in Multiple Query Optimization", in *Query Processing in Database Systems* (by W. Kim, D.S. Reiner, D.S. Batory), Springer Verlag, 1985, 191-205.

[9] W. Kim: "Global Optimization of Relational Queries: A First Step", in *Query Processing in Database Systems* (by W. Kim, D.S. Reiner, D.S. Batory), Springer Verlag, 1985, 206-216.

[10] DBS3 R.S.G. Lanzelotte and P. Valduriez: "Extending the Search Strategy in a Query Optimizer", *Proceedings 17th VLDB* 1991.

[11] A. Rosenthal, U.S. Chakravarthy, "Anatomy of a Modular Multiple Query Optimizer", *Proceedings 14th VLDB*, pp. 230-239 (1988).

[12] T.K. Sellis: "Multiple-Query Optimization", *ACM Transactions on Database Systems*, Vol. 13, 1(March 1988), 23-52.