On Saying "Enough Already!" in SQL

Michael J. Carey Donald Kossmann*
IBM Almaden Research Center
San Jose, CA 95120
carey@almaden.ibm.com, kossmann@db.fmi.uni-passau.de

Abstract

In this paper, we study a simple SQL extension that enables query writers to explicitly limit the cardinality of a query result. We examine its impact on the query optimization and run-time execution components of a relational DBMS, presenting two approaches—a Conservative approach and an Aggressive approach—to exploiting cardinality limits in relational query plans. Results obtained from an empirical study conducted using DB2 demonstrate the benefits of the SQL extension and illustrate the tradeoffs between our two approaches to implementing it.

1 Introduction

The SQL-92 query language includes support for a wide range of relational query operations, including selection, projection, many flavors of joins, unions, sorting, aggregation, grouping, and subqueries [MS93]. In addition, SQL continues to evolve, with extensions such as the object features of SQL3 and the control constructs of SQL/PSM. Surprisingly, despite its impressive query power, SQL provides no way to specify a limit on a query's result cardinality—so there is no declarative way for a SQL query writer to say "enough already!" As a result, application programs that wish to pose a query and then process at most some number (N) of its result tuples must do so, using a cursor, by submitting the entire query and then fetching only its first N results.

There are two relational database system trends that, in our view, will soon produce a strong need for declarative support for query cardinality limits. One trend is their current evolution into object-relational database systems [Sto96] and their use for managing multimedia data types such as text and images. Multimedia predicates often involve approximate matching (i.e., matching with a goodness-of-match measure), as in "show me the ten images in the database that look the most like this example" [Fag96, CG96]. This is a clear case where ranked results and cardinality limits are needed. Another trend is the increasing use of relational databases in decision-support and data warehousing environments. Ranking and cardinality limits are commonly needed by business analysts [KS95], as in "show me last week's gross rental income from the ten most popular videos."

In this paper, we extend SQL with explicit support for limiting the cardinality of a query result to a user-specified number of tuples. While a cursor-based approach is sufficient to limit the result size at the application level, in most cases it fails to limit the work done by the database engine—leading to potentially long response times, large amounts of wasted work, and a corresponding reduction in the amount of processing capacity that the database engine can make available for other, concurrent queries. After suggesting this simple SQL extension, we show how to incorporate the knowledge it provides into the query optimizer and runtime query execution subsystem of a typical relational DBMS. In addition, we present experiments using an actual relational DBMS (DB2) that quantify the potential performance gains.

The remainder of this paper is organized as follows: Section 2 presents the SQL extension and provides several examples to illustrate its usefulness. Section 3 discusses the query processing implications of the extension, proposing two implementation strategies—a conservative strategy and an aggressive strategy—for exploiting query cardinality specifications. Section 4 outlines our approach to evaluating the performance of the different strategies, which involves running benchmark queries against a synthetic database, and Section 5 applies this approach to DB2. Section 6 discusses related work, including the "optimize for early results" features found in current commercial query optimizers. Section 7 presents our conclusions and plans for future work.

2 Extending SQL

The specification of a cardinality limit for a query can be supported by extending the syntax of SQL's `SELECT` statement [C+95, CG96]. In this paper, we will do so by adding a `STOP AFTER` clause as an optional suffix, i.e.:

```
SELECT ...  FROM ...  WHERE ...
GROUP BY ...  HAVING ...
ORDER BY ⟨sort specification list⟩
STOP AFTER ⟨value expression⟩
```

An SQL `<value expression>` is any expression that evaluates to an atomic value; it may be a constant, a computation of some sort, or even a subquery [MS93]. In a `STOP AFTER` clause, the expression must evaluate to an integer value that specifies the maximum number of result tuples desired. In addition, the `<value expression>` must be *uncorrelated* with the remainder of the query in order to ensure that the query's semantics are well-defined.

*Current address: University of Passau, 94030 Passau, Germany

The semantics of a STOP AFTER query are straightforward to explain operationally: after doing everything else specified by the query, retain only the first *N* result tuples (where *N* is the integer that <value expression> evaluates to). In cases where the result of the query is totally ordered by the inclusion of an ORDER BY clause, only the first *N* tuples in this ordering are returned to the query's user or application program. In cases where the particular *N*-tuple result set is not completely defined, due to duplicate values for the ORDER BY attribute(s), the choice of which of the candidate sets of *N* tuples will be returned is left unspecified. If there is a STOP AFTER clause but no ORDER BY clause, then *any* *N* tuples that satisfy the rest of the query is considered to be a valid result. Finally, if there are fewer than *N* tuples in the result, then the STOP AFTER clause has no effect. These STOP AFTER semantics yield the same results as the cursor-based approach used by applications today.

In addition, we propose to eliminate some of SQL's restrictions concerning ORDER BY clauses. First, we suggest eliminating SQL's restriction of requiring a query's ORDER BY column(s) to also appear in its SELECT list.[1] Second, we suggest allowing ORDER BY clauses to appear in subqueries, as they become meaningful in subqueries that also have a STOP AFTER clause. Both of these extensions are trivial, and their usefulness for STOP AFTER queries should be apparent from the following examples.

2.1 Example SELECT Queries

To illustrate the usefulness of having the STOP AFTER clause, as well as illustrating its generality, let us consider a few example queries that it enables.

Example 1: Spatial Data Consider an object-relational travel database. Suppose the locations of various travel-related objects are kept track of using location attributes, which are of a user-defined Point data type, and suppose that the Point type has a *distance* function that returns the distance between two points. We can then find the five hotels closest to the O'Hare airport as follows:

```
SELECT h.name, h.address, h.phone
FROM Hotels h, Airports a
WHERE a.name = "O'Hare"
ORDER BY distance(h.location, a.location)
STOP AFTER 5;
```

Example 2: Decision-Support Data Now consider a business database containing product sales data. We can find the top 10% of the software products in terms of gross sales revenues as follows:

```
SELECT p.name, s.gross
FROM Products p, Sales s
WHERE p.type = "Software"
      AND p.prod_num = s.prod_num
ORDER BY s.gross DESC
STOP AFTER (SELECT count(*) / 10
            FROM Products p
            WHERE p.type = "Software")
```

[1] In fact, some commercial systems already support this extension.

2.2 What About Updates?

Extending the SQL SELECT statement with a STOP AFTER clause also makes it possible, without modifying SQL's INSERT, UPDATE, or DELETE statements, to express a variety of interesting updates as well. This is shown by the following example involving a database of information about baseball players.

Example 3: An Update We can give a 50% pay cut to the three worst batters as follows:

```
UPDATE Players
SET salary = 0.5 * salary
WHERE id IN (SELECT p.id
             FROM Players p
             ORDER BY p.batting_avg
             STOP AFTER 3)
```

2.3 Implications

Clearly, it is not hard to extend SQL to allow users to specify a limit on the result size of a query, and a number of interesting queries and updates become expressible. The advantage of extending SQL is that it provides information that the DBMS can exploit during query optimization and execution. The challenge, addressed in the next section, is to find ways to exploit this information—ways that are effective, yet require as few optimizer and runtime system changes as possible.

3 Processing STOP AFTER Queries

One approach to handling applications where a STOP AFTER clause is needed is to do so with little or no change to the database engine. This can be done externally, as is generally done today, or it can be done by adding a very thin additional query processing layer to the engine. This layer could strip off the STOP AFTER clause and execute it (if necessary) to compute the desired stopping cardinality; it could then submit the remainder of the query to the engine, fetch results using the system's cursor interface until the desired number of tuples have been returned to the user, and then close the cursor. As we will see later, however, both the "external" and "thin layer" approaches miss opportunities for major performance improvements that can be obtained by making use of the desired cardinality limits in the database engine.

In this section, we present a better alternative—extending the relational database engine to understand and process STOP AFTER queries. We will first describe a new query operator, called *Stop*; this operator encapsulates the STOP AFTER functionality so that other operators, such as Sort and Join, need not be changed and can be used in STOP AFTER queries just as in other SQL queries. Then, we will describe two extreme heuristics to place Stop operators in query plans: Conservative and Aggressive. At the end of the section, we will describe how Stop operators and these two heuristics can be integrated into an existing dynamic-programming based optimizer. Throughout the section, we will focus on STOP AFTER query processing in the presence of an ORDER BY clause because we expect this to be the most common case

in practice; STOP AFTER queries without an ORDER BY clause can be processed in a very similar way. Furthermore, we will focus on STOP AFTER queries with only one query block; we will explore techniques to process queries with more than one query block (e.g., queries against views or queries with subqueries) in future work.

3.1 The Stop Operator

The Stop operator is a new logical query operator; it produces, in order, the *top* or *bottom* N tuples of its input stream. The Stop operator requires three parameters to be provided at query initialization time. The first is **N**, the desired number of result tuples. The second is a **Sort Directive**, which will be one of three values: *desc*, *asc*, or *none*. If the sort directive is *desc* (*asc*), the Stop operator sorts its input stream and returns the *top* (*bottom*) N tuples in descending (ascending) order. If the sort directive is *none*, the Stop operator simply returns the first N tuples from its input stream; the *none* option is chosen by the optimizer when the Stop operator's input stream is known to already be appropriately sorted. The third parameter to Stop is a **Sort Expression**. If the sort directive is *desc* or *asc*, the Stop operator sorts its input according to this sort expression, which is usually identical to the ordering expression from the ORDER BY clause of the query.

Like other logical operators (e.g., Join), the Stop operator can have more than one physical operator that is capable of implementing it in a query plan. Clearly, the implementation of the Stop operator should at least be dependent on its sort directive. Accordingly, we define two different physical Stop operators here: *Scan-Stop*, for when the sort directive is *none*, and *Sort-Stop*, for when the sort directive is *desc* or *asc*. We now discuss a possible implementation and a cost model (for the optimizer's use) for each one.

3.1.1 Scan-Stop

The Scan-Stop operator is extremely simple. Scan-Stop is a pipelined operator that simply requests and then passes each of the first N tuples of its input stream on to its consumer (i.e., to the operator above it in the query plan), after which it closes down its input stream and returns an end-of-stream indicator to its consumer. As a result, the cost of the Scan-Stop operator itself is negligible, and the total cost of a query subplan rooted at a Scan-Stop operator is dominated by the cost required to produce the first N tuples of its input stream. In a state-of-the-art relational DBMS, the query optimizer's cost model provides estimates for the total cardinality of a plan's output (ALL), the cost to produce the first tuple of a plan's output ($\text{cost}_p(1)$), and the cost to produce ALL output tuples ($\text{cost}_p(\text{ALL})$). Given estimates for these quantities for the subplan that generates the input stream for the Scan-Stop operator, the optimizer can estimate the cost, $\text{cost}_s(N)$, for the whole plan rooted at the Scan-Stop(N) operator as follows:

$$\text{cost}_s(N) = \text{cost}_p(1) + (\text{cost}_p(\text{ALL}) - \text{cost}_p(1)) * \frac{N}{\text{ALL}} \quad (1)$$

This estimate assumes that the tuples after the first one are generated smoothly by the subplan that feeds into the Scan-Stop operator, i.e., that $\text{cost}_p(k+1) - \text{cost}_p(k)$ is more or less constant for $1 \leq k < \text{ALL}$. Also, it assumes that $N \leq \text{ALL}$ (since for $N > \text{ALL}$, $\text{cost}_s(N) = \text{cost}_s(\text{ALL})$).

The nature of Equation 1 implies that the optimizer is likely to favor pipelined query plans (e.g., query plans with nested-loop join operators) for STOP AFTER queries, particularly when the cardinality limit N is small. This is because pipelined execution plans quickly produce their first row (i.e., $\text{cost}_p(1) \approx 0$); therefore, the relative cost of the STOP AFTER query will be proportional to its cardinality limit N. In contrast, if the Scan-Stop operator's input stream is produced by a pipeline-breaking subplan (e.g., one with a sort-merge join in it), then the cost to produce N tuples will be almost the same as the cost to produce ALL tuples.

3.1.2 Sort-Stop

If the input of a (logical) Stop operator is not already sorted, then the Stop operator must consume and sort its whole input to produce the *top* or *bottom* N output tuples. For relatively small N (which we expect to be the most common case), the sorting can be carried out in main memory using a priority heap [Knu73]. The first N tuples consumed by the Sort-Stop operator can be inserted into a priority heap, and the remaining tuples can then be tested with the heap's membership bound to determine whether or not the new tuple's value warrants its insertion into the heap. The cost of producing N results with this implementation of the Sort-Stop operator has the following three components: (1) the cost of generating the whole input stream for the Sort-Stop operator ($\text{cost}_p(\text{ALL})$), (2) the cost of testing $\text{ALL}-N$ tuples against the heap's membership bounds ($(\text{ALL} - N) * C$, where C is the cost of a comparison), and (3) the cost of inserting i qualifying tuples into a heap with at most N elements (where i is estimated below). In all, the resulting plan cost can be estimated as shown in Equation 2; again, we assume $N \leq \text{ALL}$, as for $N > \text{ALL}$, $\text{cost}_s(N) = \text{cost}_s(\text{ALL})$.

$$\text{cost}_s(N) = \text{cost}_p(\text{ALL}) + (\text{ALL} - N) * C + i * \log(N) \quad (2)$$

Assuming randomly ordered data, i can be estimated as:

$$i = N + \frac{N}{N+1} + \frac{N}{N+2} + \ldots + \frac{N}{\text{ALL}} = (\mathbf{H}_{\text{ALL}} - \mathbf{H}_N + 1) * N \quad (3)$$

(Here, $\mathbf{H}_k$ denotes the k-th harmonic number [Knu73].)

Again, this cost estimate assumes that N is small and that a heap of N tuples will fit in memory. For larger N, external sorting is required. Although Sort-Stop's specific requirements invite the design of a specialized external Sort-Stop algorithm, we will not discuss such an algorithm here. Instead, we will assume a Sort-Stop implementation that uses an ordinary external Sort operator in conjunction with the Scan-Stop operator when N is large.

3.2 The Conservative Policy

We now turn to the question of where Stop operators should be placed in a query plan. Stop operator placement involves a fundamental tradeoff: On one hand, a deeply placed Stop operator will cut down the size of intermediate results early, thus reducing the cost of operators higher up in the plan. On the other hand, it is possible for a deeply placed Stop operator to eliminate too many tuples of an intermediate result; this situation must then be trapped at run-time, and the execution of the query must be *restarted* to produce the remaining (missing) tuples of the query result.

An ideal optimizer would weigh the risk and cost of restarts against the benefits of deep Stop operators placement. However, current optimizers have no notion of risk. We therefore propose the use of a heuristic *Stop operator placement policy* to assist the optimizer in placing Stop operators in a plan. In this subsection, we will present the *Conservative* policy, which completely avoids restarts by restricting Stop operator placement to "provably safe" locations in a plan. In the next subsection, we will present the *Aggressive* policy, which does permit the placement of Stop operators at unsafe places.

3.2.1 Conservative Stop Placement

The Conservative policy introduces Stop operations as early as possible in a plan, subject to the following principle which makes sure that under no circumstances restarts are required:

> *Conservative Stop Placement Principle:* Never insert a Stop operator at a point in a plan where its presence can cause tuples to be discarded that may be required to compose the requested *N* tuples of the query result.

Intuitively, a Stop operator can be applied to a tuple stream if every tuple in that stream is guaranteed to generate at least one tuple of the overall query result. During query optimization, this condition can be tested for a tuple stream produced by a subplan by (i) inspecting the query predicates that remain to be applied following the subplan (to complete the query), and (ii) examining the integrity constraints involving columns that participate in these remaining predicates. The condition is satisfied for the subplan's output stream if each of the remaining predicates is *non-reductive*, defined as follows:

1. the predicate is of the form $x = y$, where x is an expression computable from the stream and y is an expression involving one or more tables yet to be added to the plan, and

2. it can be inferred from the database integrity constraints that (i) x cannot be null, and (ii) for each x there must exist at least one y such that $x = y$ holds.

For example, if x is a column with a `NOT NULL` constraint, and the database has a referential integrity constraint declaring that x is a foreign key that references a table yet to be joined whose key is y, then these criteria are met—making $x = y$ a non-reductive predicate. In the case of outer-join predicates, the two conditions can be relaxed because a tuple from the outer relation survives even if no matching tuple from the inner relation exists. In the case of `GROUP BY` or `DISTINCT` queries, if the grouping operation (or duplicate elimination) remains to be done, a similar set of rules can be derived. In this case, the tuple stream must be functionally dependent on the `GROUP BY` (or `DISTINCT`) column(s), and any `HAVING` predicates must also satisfy the conditions above so that all groups will survive. Naturally, the Conservative Stop insertion conditions will always hold at the root of a plan for the whole query because no predicates remain to be applied at that point. In all cases, the Stop operator inserted will have a cardinality parameter of N, where N is the value specified in the `STOP AFTER` clause.

3.2.2 Conservative Plan Examples

Consider the following database for managing a company's employees, departments, and employees' travel expense accounts (TEAs):

> Emp(empId, name, salary, *works_in*, *teaNo*)
> Dept(dno, name, budget, function, description)
> TEA(accountNo, expenses, comments)

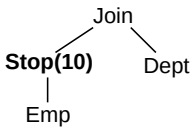
The underlined columns are the primary keys for the tables, while the italicized columns (`works_in, teaNo`) are foreign keys. In addition to key and referential integrity constraints, we assume that the database has an integrity constraint to enforce the fact that every employee must work in a department; i.e., `Emp.works_in` is `NOT NULL`. Finally, we assume that not all employees travel, so the corresponding specification for `Emp.teaNo` would say `NULL ALLOWED`.

Figure 1 shows three `STOP AFTER` queries, with corresponding query execution plans, to show how Conservative Stop placement works. In Query 1, a Stop operator can be placed beneath the join on the `Emp` side (i.e., just above a scan of the `Emp` table). This is permitted because `e.works_in = d.dno` is a non-reductive predicate. In contrast, in Query 2, the placement of a Stop operator will be forbidden at that point in the plan, as two predicates remain: `e.works_in = d.dno` and `d.function = 'Research'`. The latter predicate is *not* non-reductive: If the `Emp` tuple stream were reduced to the 10 highest paid employees before the join, the query could come up short, as joining those employees with only research departments could eliminate many of them.[2] Finally, in Query 3, which asks for the bottom 10 tuples by departmental budget rather than the top 10 by employee salary, the Stop operator must again be placed above, rather than below, the join. In this case, the join predicate itself is non-reductive, as our integrity constraints provide no guarantee that every department has at least one employee.

To complete our discussion of the Conservative approach, we need to raise one subtle point that we have glossed over up to now. Even if a join predicate satisfies the criteria for being non-reductive, it is possible that the join may reduce the

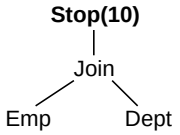
[2] Lou Gerstner, for example, is not an IBM researcher!

```
SELECT *
FROM Emp e, Dept d
WHERE e.works_in = d.dno
ORDER BY e.salary DESC
STOP AFTER 10;
```



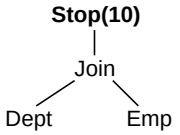
Query 1 & Plan 1

```
SELECT *
FROM Emp e, Dept d
WHERE e.works_in = d.dno
AND d.function = 'Research'
ORDER BY e.salary DESC
STOP AFTER 10;
```



Query 2 & Plan 2

```
SELECT *
FROM Emp e, Dept d
WHERE e.works_in = d.dno
ORDER BY d.budget
STOP AFTER 10;
```



Query 3 & Plan 3

Figure 1: Example Queries and Conservative Plans

cardinality of the input stream in question—this can happen if, and only if, the to-be-joined table is empty, in which case even a Cartesian product with this table will reduce the stream's cardinality (to zero). While true, this does *not* invalidate our specification of the Conservative approach, as the query result is itself empty in this case—so pruning the tuple stream will not cause required tuples to be erroneously discarded.

3.3 The Aggressive Policy

The Conservative policy has several advantages: its Stop operators simply take their value of N from the `STOP AFTER` clause, and Conservative execution plans are a straightforward extension of regular plans (with Stops added in strategic but safe places). It also has a disadvantage, however: it only inserts Stop operations at points where all remaining predicates are non-reductive, so in the absence of `NOT NULL` and referential integrity constraints, it cannot place Stop operators below joins in multi-way join plans. Given that joins are expensive operations that benefit significantly from input cardinality reductions, it would be nice if we could relax this restriction. That is the goal of the *Aggressive* policy.

3.3.1 Aggressive Stop Placement

The Aggressive policy seeks to introduce Stop operations as early in query plans as possible, regardless of whether or not their insertion is provably safe. In other words, the Aggressive policy operates according to the following principle:

> *Aggressive Stop Placement Principle:* Insert Stop operators in query plans wherever they can provide a beneficial cardinality reduction.

Thus, an optimizer that employs the Aggressive policy will consider a Stop insertion at each point in a query plan tree where the first (high order) column of the query's `ORDER BY` clause can be computed, i.e., at each point where a set of *top* (or *bottom*) tuples can be identified with respect to the requested result order. Several new issues arise when the optimizer is able to generate plans with a Stop operator below one or more reductive predicates. First of all, the optimizer must now compute the stopping cardinality for such a Stop operator; this can be done using a simple formula based on the cardinality estimates utilized by a typical state-of-the-art query optimizer. Let $\mathrm{ALL}_{\mathrm{query}}$ be the estimated number of tuples that would be produced by the query as a whole *without* any Stops, and let $\mathrm{ALL}_{\mathrm{subplan}}$ be the estimated number of tuples produced by the subplan that feeds tuples into the Stop operator in question. As always, let N denote the target cardinality of the query as a whole (the value specified in the `STOP AFTER` clause). Given these quantities, the stopping cardinality for the Stop operator in question should be set to:

$$N_{\mathrm{Stop}} = \frac{\mathrm{ALL}_{\mathrm{subplan}}}{\mathrm{ALL}_{\mathrm{query}}} * N \tag{4}$$

In other words, the Stop should reduce the cardinality of the tuple stream at its point in the plan in proportion to the optimizer's estimate of the overall cardinality reduction requested by the user. In practice, it would be better to inflate the stopping cardinality by, say, 20% to accommodate estimation error; we will explore this issue further in Section 5.

Since setting the stopping cardinality in this way relies on cardinality estimates, additional steps must be taken to ensure that the right number of result tuples is ultimately produced by an Aggressive query plan. To handle situations where the Aggressive optimizer sets a Stop operator's stopping cardinality too high, another (final) Stop operator must be placed at (or near) the root of the plan. To handle situations where the stopping cardinality is set too low, the query plan must be *restarted* to produce the missing tuples. We propose to handle the required restart activity at runtime by having a well-placed *Restart* operator in the Aggressive query plan. We present this operator in the following subsection, showing how it can be added to an Aggressive Stop plan as part of a plan refinement step that takes place after Aggressive query optimization is completed.

3.3.2 Restarts and Plan Refinement

The job of a Restart operator is to make sure that, above the point where it is placed in the query plan, at least N tuples will surely be generated. If the Restart operator's input stream is exhausted before N tuples are received, then it must "restart" the query subplan beneath it to generate the required, but missing, results. There are several possible strategies for actually restarting a query subplan. Ideally, we would like to compute the missing tuples without recomputing any of the results that we have already gathered. However, this would in some situations be difficult (if not impossible) to accomplish

without making all of the system's operators restartable.

To handle restarts without changing existing runtime operators, we adopt a simple approach: the Restart operator will simply close and then re-open the iterator associated with the operator that feeds it tuples from the subplan immediately beneath it—causing the root operator of the subplan to recursively close and re-open its own input iterator(s), and so on down the line. The Stop operator in the scope of the subplan will reconfigure itself when it is reopened by increasing its stopping cardinality by a predetermined factor (e.g., a factor of 2) to let more data through the next time around. The other operators in the subplan will simply start over from the beginning, though their cost will be lower since much of the relevant data will still be in the buffer pool. To ensure that the correct number of results are ultimately produced, the Restart operator will repeat this process (if necessary) until it has produced N tuples or there are no more tuples to produce.

Restart operator insertion can be done by post-processing the optimizer's chosen query plan as follows: Find the non-Conservative Stop operator in the plan and insert a Restart operator at the first point above it where all remaining predicates are non-reductive. If we produce N results there, we are "safe"; no further cardinality shrinkage can occur. The post-processor can simply walk up the plan tree from the non-Conservative Stop operator in order to place the Restart as deeply as possible (to minimize the cost of a restart).

For Aggressive query plans that are fully pipelined, or have pipelined sections, there is an additional plan refinement that is possible and very useful: *Scan-Stop pullup*. This transformation pulls Scan-Stop operators residing in a pipelined segment of a query plan up to the top of the pipeline. This reduces the *risk* of restarts, as it re-positions the Scan-Stop above reductive predicates in the pipeline (whose impact on the tuple stream cardinality might be difficult to estimate). In the extreme case, when all the remaining reductive predicates are applied in the pipeline, Scan-Stop pullup converts an Aggressive Scan-Stop operator into a Conservative one—completely eliminating the risk of restarts; Scan-Stop pullup must, therefore, be performed before Restart operator placement, and when the pullup transformation is performed, the cardinality parameter of the affected Scan-Stop operator must, of course, be recomputed based on its new position in the query plan. Note that Scan-Stop pullup is never detrimental. Consider, for example, a Scan-Stop operator sitting immediately beneath a pipelined operator such as a nested loop index join (NLIJ). In such a case, nothing is gained by having the Scan-Stop operator sit beneath the NLIJ operator, because the NLIJ produces output incrementally, on demand—i.e., its cost is proportional to the amount of data that it is asked to materialize, not to the size of its inputs. We can thus pull the Scan-Stop operator up above the NLIJ without increasing the cost of the plan. This pullup opportunity arises because the two NLIJ plans will have the same cost during plan enumeration, and the risk of restarts is not taken into account during enumeration.

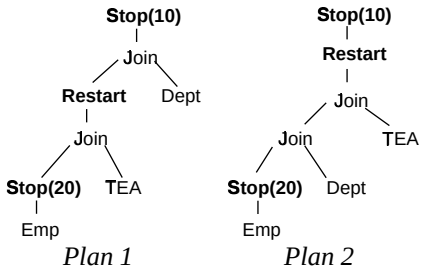


Figure 2: Aggressive Stop Push-Down for a 3-Way Join

3.3.3 Aggressive Plan Examples

Figure 2 shows two examples of query plans that could be produced by Aggressive Stop optimization followed by plan refinement for the following query:

```
SELECT e.name, e.salary, d.name, t.expenses
FROM Emp e, Dept d, TEA t
WHERE e.works_in = d.dno
      AND e.teaNo = t.accountNo
ORDER BY e.salary DESC;
STOP AFTER N;
```

Recall that all employees work in some department, but not every employee has a travel account. For illustration, suppose that half of the employees have a travel account (i.e., an associated TEA tuple). In Plan 1 of Figure 3, where Emp is joined with TEA first, the Aggressive approach is able to place a Stop operator just above the scan of Emp, at which point the stopping cardinality is set to 20 (because $\frac{\mathrm{ALL}_{\mathrm{subplan}}}{\mathrm{ALL}_{\mathrm{query}}} = 2$ and $N = 10$). Because this Stop operator is non-Conservative, a corresponding Restart operator is needed; it goes above the first Join, where the Conservative approach would have placed a Stop(10) operator. If the join is non-pipelined (e.g., a sort-merge or hash join), the Aggressive plan can provide a significant join cost savings over its Conservative counterpart.

In Plan 2, the join with the TEA table comes last. Since its join predicate is reductive, the Conservative plan for this join order would only be able to place a Stop(10) operator at the root of the plan. In contrast, the Aggressive approach again places a Stop(20) operator right above where Emp is accessed. The potential cost savings in this case is obviously large—if either join is non-pipelined, Plan 2 is quite a bit cheaper than its Conservative (in this case "do nothing") counterpart.

3.4 Extended Query Optimization

Stop operators impact the cost of other operations in a query plan, thus affecting optimization decisions such as access path selection, join methods, and join orderings. We now show how to extend a modern query optimizer—one that uses dynamic programming a la [S+79]—to incorporate our Stop placement policies.

3.4.1 Plan Enumeration

Modern query optimizers use rules to control the enumeration of alternative query plans [GD87, Loh88]. To enumerate all possible plans with Stop operators, we propose modifying the optimizer's rule set to model the Stop operator as another

kind of access path for intermediate results. This will cause the query optimizer to consider adding a Stop operator immediately after each of its usual ways of accessing either base data or subplan data. Moreover, this change introduces Stops with minimal impact on the optimizer as a whole, as the rest of its rule set remains untouched. This is important to ensure modularity, and therefore maintainability, of the Stop-enhanced rule set. The insertion of a Stop operator must also be considered at the root of each query plan to ensure that the right number of result tuples (as specified by the `STOP AFTER` clause) will be returned to the application. This can be accomplished through another new rule that considers adding a Stop operator at this final point.

Once these additional Stop rules have been established, both the Conservative and Aggressive policies can easily be integrated by defining conditions that control in which situations (i.e., for which subplans) a Stop rule can be applied; again, other rules of the optimizer are not affected. A vital condition for both policies is that a Stop operator may only be generated at places in a plan where the *top* or *bottom* tuples can be identified (i.e., where at least the first column of the query's `ORDER BY` expression is computable). To implement the Conservative policy, the conditions of the Stop rules would further be restricted to generate Stop operators only at safe places (as defined in Section 3.2). Such a restriction would not be considered for the Aggressive policy.

An important issue to make plan enumeration more efficient is to avoid the enumeration of plans with superfluous Stop operators. To keep track of the places where a Stop operator is useful, we introduce an additional plan property called *stopping cardinality*. (The use of plan properties is a common technique implemented in most query optimizers [GD87, Loh88, S+96].) For the Conservative policy, the stopping cardinality property can take one of two values: ENFORCED, indicating that the plan's result stream already satisfies the `STOP AFTER` clause's cardinality specification, or NOT ENFORCED, indicating that this is not true. At leaves of plans (e.g., table scans), the stopping cardinality property is set to NOT ENFORCED; it is set to ENFORCED when a (Conservative) Stop operator is added to the plan. This property is "destroyed" (reset to NOT ENFORCED) by operators that potentially change the cardinality of their input streams (e.g., certain joins), and is transported by other operators. Using the Conservative policy, therefore, Stop operators can only be added to NOT ENFORCED subplans, and again, this constraint can be implemented in the condition part of the rules.

In addition to ENFORCED and NOT ENFORCED, the stopping cardinality can be set to ASSISTED to support the Aggressive policy. This is done by a non-Conservative Stop operator whose stopping cardinality is based on cardinality estimates, and it specifies that cardinality reduction has been performed but that an additional Stop operator is needed at (or near) the root of the plan to guarantee ENFORCED cardinality for the whole plan. An ASSISTED stopping cardinality is transported by every operator; it is overwritten by the final Conservative Stop operator, so only plans with at most one non-Conservative Stop operator are enumerated.

3.4.2 Impact on Pruning

In the dynamic programming approach to query optimization, a (sub-) plan P_1 is pruned in favor of a (sub-) plan P_2 if and only if P_1 has both a higher cost and weaker properties than P_2. In addition to the properties considered in a traditional optimizer, our `STOP-AFTER`-extended optimizer must respect certain additional pruning conditions. If the output of P_1 and P_2 is ordered according to the query's `ORDER BY` expression, then P_1 may only be pruned in favor of P_2 if P_1 has higher cost and weaker properties (as in traditional optimizers) *and* cost(Scan-Stop(P_1, N_{stop})) $\geq$ cost(Scan-Stop(P_2, N_{stop})).[3] Otherwise, even if P_1 has higher cost than P_2 for producing all results, P_1 may be part of the overall best plan to produce N results. As stated in Section 3.1.1, this condition will force the optimizer to retain plans with (seemingly expensive) pipelined operators, such as scans of unclustered indexes.

If P_1 and P_2 are not ordered according to the query's `ORDER BY` expression, all of their tuples must be produced to compute the overall ordered query result. In this case, our Stop-enhanced optimizer must still respect certain additional pruning conditions to retain query plans with deep Stop operators. To see why, consider the following query:

```
SELECT *
FROM Emp e, Dept d
WHERE e.works_in = d.dno
ORDER BY e.salary
STOP AFTER 100
```

Suppose that the database has 10,000 employees and 1,000 departments, and that a Sort-Stop operation on 10,000 tuples costs 40,000 units, as compared to 200 units for either a Sort-Stop or a plain Sort operation on 100 tuples. Also, assume that a sort-merge join costs 43,000 units to join 10,000 employees with 1,000 departments, versus 3,200 units for a sort-merge join between 100 employees and 1,000 departments. (For simplicity, we use $n \log n$ to approximate the cost of sorting n tuples for a Sort-Stop or sort-merge join.)

During plan enumeration, the following two subplans will be enumerated, among others:

P_1 = SMJoin(Sort-Stop(Emp), Dept)
P_2 = SMJoin(Emp, Dept)

Given our assumptions, the cost estimate for subplan P_1 is 43,200 units (40,000 + 3,200), while the cost of P_2 will be estimated as 43,000 units, which is slightly lower. To produce the final query result in the requested order (`e.salary`), a Sort operator costing 200 units can be attached to P_1, giving a total cost of 43,400 units. For P_2, a Sort-Stop operator costing 40,000 units is required, resulting in a total cost of 83,000 units. Thus, the winning plan is based on P_1 rather than P_2—even though P_1 itself has higher cost than P_2. This is because P_1 has stronger *cardinality* properties; i.e., P_1 produces 100

[3] Scan-Stop cost as in Equation (1); N_{stop} as in Equation (4.

```
SELECT e.name, e.salary
FROM Emp e
ORDER BY e.salary DESC
STOP AFTER N;
```

Test Query 1

```
SELECT e.name, e.salary,
       d.name
FROM Emp e, Dept d
WHERE e.works_in = d.dno
ORDER BY e.salary DESC
STOP AFTER N;
```

Test Query 2

```
SELECT e.name, e.salary,
       d.name, t.expenses
FROM Emp e, Dept d, TEA t
WHERE e.works_in = d.dno
      AND e.teaNo = t.accountNo
ORDER BY e.salary DESC
STOP AFTER N;
```

Test Query 3

Figure 3: STOP AFTER Test Queries

tuples and has a stopping cardinality of ENFORCED, whereas P_2 produces 10,000 tuples and has a stopping cardinality of NOT ENFORCED. In general, a plan has stronger *cardinality* properties than another plan and may not be pruned if it has a smaller estimated result cardinality or a stronger stopping cardinality property obeying the total order: ENFORCED > ASSISTED > NOT ENFORCED.

Obviously, adding rules for Stop enumeration and restricting pruning increases the sizes of the query optimizer's search space and its set of candidate query plans. Studying this impact is beyond the scope of this paper, but given the efficiency and pre-existing complexity of modern query optimizers, we do not believe that our extensions will seriously impact their performance. In fact, related search space issues have been studied in the context of GROUP BY query optimization (e.g, [CS94]); those results are directly applicable here.

4 Evaluation Methodology

We have proposed two approaches to placing Stop operators in execution plans for STOP AFTER queries. To compare them to each other, and to the "do nothing" approach, we will employ a synthetic database and several representative STOP AFTER queries. Our approach will be to "cheat"—using our knowledge of the attribute value distributions to translate the STOP AFTER queries into regular SQL queries with additional predicates that force the target DBMS do an amount of work that is essentially the same as if its query plans were Stop-enhanced.

4.1 Synthetic Database

For the test database, we used the tables and integrity constraints presented in the examples of Section 3. We controlled the attribute value distributions to facilitate our test queries and built indexes on all the relevant attributes.

Our test database consists of the Emp, Dept, and TEA tables introduced earlier. The integrity constraints specify that every employee works in a department, but not every employee has a travel account. Each table holds 100,000 rows of 100 bytes each. (We also tried other table sizes, including a much smaller Dept table, but will not show those results here; we saw large performance benefits for our STOP AFTER schemes, but since the Dept join itself was inexpensive, the basic results were similar to our single-table query results.) The tables' attribute values were synthesized to control range predicate selectivities and join selectivities. Emp.salary values ranged from 0 to 99,999, and the key and foreign key values were chosen so that each Emp row joins with exactly one Dept row and one TEA row. We created clustered indexes on Dept.dno and TEA.accountNo to support joins between each of these tables and the Emp table. Our experiments varied the index created on Emp.salary, which was the field used in the ORDER BY clauses of our test queries—we tried clustered, unclustered, and no index at all, using a descending order index in the cases where this field was indexed.

4.2 Test Queries

Our test queries are shown in Figure 3. The queries range from a simple single-table query to more complex join queries. The first query simply finds the N most highly paid employees. Its purpose is to investigate the impact of the STOP AFTER clause on (1) row blocking (for sending the result set back to the client), (2) access path selection, and (3) costs to sort query results. The second query finds the N most highly paid employees and their department names. Its purpose is to show the impact of the STOP AFTER clause on the choice of join method. The third query finds the N most highly paid employees, including both their department names and their travel expenses. Its purpose is to provide a more complex join example to explore the tradeoffs between the Conservative and Aggressive approaches. For each query, we varied the requested stopping cardinality N from 1 to 100,000.

4.3 DB2 Simulation Approach

Given these queries, we wanted to know—before embarking on an actual implementation, and for a real DBMS with sophisticated query optimization (including cardinality estimation) and an industrial-strength runtime system—how much benefit could be obtained from our STOP AFTER optimizations as a function of the stopping cardinality N and the physical database design. We chose to "simulate" the execution of Stop plans on IBM's DB2 for Common Servers by replacing each query's STOP AFTER clause with an appropriately designed Emp.salary predicate.

For Queries 1 and 2, we can create regular DB2 queries that require the same amount of work by dropping their STOP AFTER clauses and adding a predicate of the form e.salary >= 100,000-N. Rewriting the queries this way simulates the effect of placing a Stop operator just above the table or index scan of the table Emp, as it gives DB2 a predicate to apply at that point that has the same filtering effect as the Stop operator would have there.

For Query 3, we can simulate a Conservative version of Plan 1 of Figure 2, where there is one Stop operator that sits immediately above the join Emp $\bowtie$ TEA (in place of Figure 2's Restart operator), via the following DB2 query:

```
SELECT e.name, e.salary, d.name, t.expenses
FROM Emp e, Dept d, TEA t
WHERE e.works_in = d.dno
      AND e.teaNo = t.accountNo
      AND e.salary + t.zero ≥ 100,000-N
ORDER BY e.salary DESC;
```

The field `TEA.zero` is a special field of the TEA table that simply contains the value 0 in each row. Using this field, we have translated the `STOP AFTER N` clause in the original version of Query 3 into the predicate `e.salary + t.zero >= 100,000-N`; doing so provides the same selectivity as the simpler predicate introduced for Queries 1 and 2, but forces DB2 to wait until after `Emp` and `TEA` have been joined before restricting the number of tuples (where the Conservatively placed Stop operator would do it).

We can also simulate an Aggressive version of Query 3, one where the Stop operator is pushed down to the `Emp` table, and where there is a Restart operator at the top of the query plan to handle the case where an estimation error causes the plan to prematurely run out of tuples.[4] We can do so by running the following query, which has a pair of parameterized predicates on `Emp.salary`, against DB2 from within a C++ application program that uses the parameters to simulate the effects of estimation and restarts:

```
SELECT e.name, e.salary, d.name, t.expenses
FROM Emp e, Dept d, TEA t
WHERE e.works_in = d.dno
      AND e.teaNo = t.accountNo
      AND ? ≤ e.salary AND e.salary < ?
      AND e.salary + t.zero ≥ 100,000-N
ORDER BY e.salary DESC;
```

Instantiation of the two query parameters (the "?"s) enables the C++ program to simulate underestimation, precise estimation, and overestimation of the required cardinality for the Stop operator; it also allows the program to simulate a restart of the query. In addition to this parameterized range predicate, the DB2-translated aggressive query still includes the special predicate involving `TEA.zero` to simulate the presence of a Stop operator after Emp ⋈ TEA; this operator's job is to cut down the cardinality when the parameters of the range predicate lead to a range that is too large (simulating a case where the required cardinality is overestimated).

4.4 Test Environment Details

Our tests were run on an IBM RS/6000 PowerStation 550 with 128MB of main memory running AIX 4.1.4. We used the product version of IBM's DB2 for Common Servers (Version 2.1.1). We used DB2's default settings almost exclusively, which gave us 4 MB of main-memory buffer space (desirable so that we didn't have to generate a huge database for our tests), a 512 KB application heap for row blocking, and query optimization level 5. All queries were run against a warm database (though this was largely a non-issue because the database size was much larger than the buffer pool). We measured warm times because cold query execution times were not accurately measurable for the shorter queries (such as index scans for small values of N); obtaining accurate times required running them several times and reporting (*total_elapsed_time* / *no_of_runs*).

[4] Note: We could not simulate a Restart operator in the middle of a plan.

5 Experiments and Results

In this section, we use our synthetic database and test queries to explore, using the DB2 "simulation" scheme outlined in the previous section, (i) the performance benefits of our `STOP AFTER` optimization techniques and (ii) the tradeoffs between Aggressive and Conservative query plans.

5.1 Traditional vs. STOP AFTER Query Optimization

We begin by examining the benefits of `STOP AFTER` query optimization for single table queries; we then explore its impact on join queries. Results are given for three approaches in this section: (1) TRADITIONAL, which is the traditional, "do nothing" approach in which an application program hands DB2 the original `STOP AFTER` query with the `STOP AFTER N` clause removed, opens a cursor over the results, requests N rows (one at a time), and then closes the cursor. (2) TRAD(NRB), which is the traditional approach, but with no row blocking. By default, the DB2 server process ships answer sets to client processes in large blocks of rows (based on the application heap size) to minimize communication costs; the TRAD(NRB) results were obtained by turning this feature off, thereby making the server send results back one row at a time. (3) STOP-AFTER which is the Conservative `STOP AFTER` optimization approach; these results were obtained by sending DB2 the "simulated" `STOP AFTER` queries described in Section 4.3.

5.1.1 Single Table Queries (Query 1)

Figure 4 presents the performance results for Query 1 as a function of N, the query's stopping cardinality, when there is a clustered index available on `Emp.salary`. Query 1 gets the names and salaries of the N most highly paid employees. In all three approaches, DB2 utilizes the clustered index to process the query—to extract the results in the desired order without sorting—so all of the performance differences seen here are due to row blocking. TRADITIONAL has very poor performance (note the logarithmic y-axis!) for small N because it groups query results into blocks of about 500 rows before sending them back to the client; this is clearly a waste when relatively few of the result rows are ultimately consumed by the application. TRAD(NRB) performs well for small N, but becomes relatively worse for large N because it requires a client/server interaction for each result row. At $N = 100,000$, TRAD(NRB) ends up being 2.5 times slower than TRADITIONAL. In contrast, STOP-AFTER does well throughout the entire N range. In this case, DB2 sends result rows back in chunks of min(N, application_heap), as the STOP-AFTER plan limits the result size on the server side; this is good for small N because it avoids wasted work, and good for large N because it involves low overhead when many results are indeed desired by the application program.

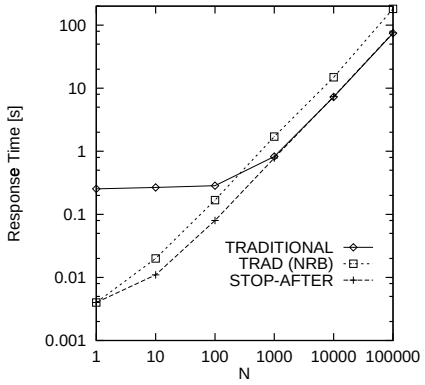


Figure 4: Resp. Time (log), Q1
Clustered Index on Emp.salary.

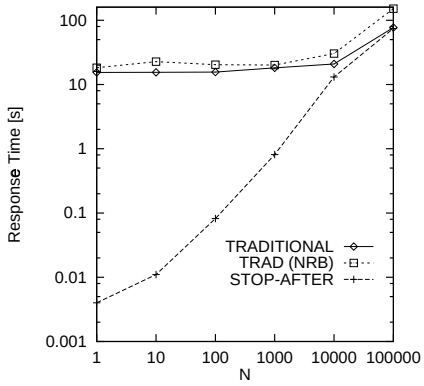


Figure 5: Resp. Time (log), Q1
Unclustered Index on Emp.salary.

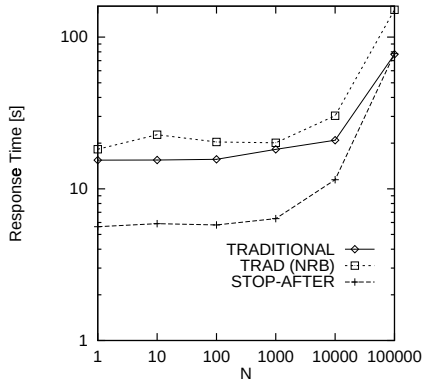


Figure 6: Resp. Time (log), Q1
No Index on Emp.salary.

Figure 5 presents the results for Query 1 when there is an *unclustered* index on `Emp.salary`. The traditional approaches use a table-scan followed by a sort to obtain the query result; they do not know that the application program will only request N of the results, and the table-scan/sort plan is a cheaper plan for Query 1 in the absence of the `STOP AFTER` clause. In contrast, the STOP-AFTER approach chooses the unclustered index to obtain the results when N is small, as it has enough information to compare the costs of producing the requested N tuples each way. As a result, STOP-AFTER performs much better than either of the traditional approaches for most of the range of N values, entering the ballpark of their performance only for $N >= 10,000$. TRADITIONAL and TRAD(NRB) perform similarly until this region, at which point the cost of client/server communication becomes significant enough to cause TRAD(NRB) to perform worse than TRADITIONAL.

Lastly, Figure 6 presents the results for Query 1 when there is no index on `Emp.salary`. In this case, all three approaches sort to produce the query's ordered result set. STOP-AFTER has the lowest cost here (by about a factor of three at $N = 1$) because it only sorts the N rows actually of interest to the application program.[5] In contrast, both traditional approaches sort 100,000 rows, making them quite a bit more expensive. Again, TRAD(NRB) is somewhat slower than TRADITIONAL due to the cost of shipping the query result back to the application one row at a time. (In fact, from this point on we will stop showing TRAD(NRB) results, as it consistently lost to TRADITIONAL in the remaining tests.)

5.1.2 Join Queries (Query 2)

Figures 7- 9 show the results for Query 2. This query asks for the name, salary, and department name of the N most highly paid employees, so it is like Query 1 with the addition of a `Dept` join. Regardless of the availability or type of `Emp.salary` index, STOP-AFTER performs much better than TRADITIONAL. The reason for this is quite simple: Since the STOP-AFTER plan is chosen based on the value of N, and the join predicate is non-reductive (given the test database's integrity constraints), only the first N rows of `Emp` are joined with `Dept` in this plan. Moreover, when N is small, the STOP-AFTER query plan uses the nested-loop index join method to perform the join, which is cheaper than the sort-merge join method used in the TRADITIONAL (N-insensitive) plan. Sort-merge is the superior strategy for a full Emp $\bowtie$ Dept join, but not for a restricted join; although the `Dept` table can be accessed in `dno` order using its clustered index, the sort-merge query plan still involves sorting the `Emp` table on the `works_in` column as well as sorting the join's result to produce the results in `salary` order. The advantage of the STOP-AFTER approach is especially pronounced in Figures 7 and 8. Here, the STOP-AFTER plan uses the `Emp.salary` index to cheaply obtain the most highly paid employees (by piping the results of an index scan into an inexpensive Scan-Stop operator); moreover, for small N, where it uses the nested-loop index join method to probe the `Dept` table, no further sorting is needed.

In addition to these results, we also ran tests with no index on `Dept.dno`. We observed smaller differences in those tests, but STOP-AFTER still significantly outperformed TRADITIONAL. STOP-AFTER was better for small N because its ability to do Stop push-down led to cheaper joins as well as cheaper sorts; for $N = 10$, for example, STOP-AFTER outperformed TRADITIONAL by a factor of two when an index existed on `Emp.salary`, and it won by a factor of 1.5 without this index.

5.2 Aggressive Plans: Benefits and Risks

We have seen that `STOP AFTER` query plans can provide a large cost savings with respect to traditional query plans with cursors. But what about Aggressive plans—how much additional savings can they provide? And how costly are the estimation errors that may occur? These questions are the focus of our next tests which are based on Query 3, the 3-way join query that asks for the name, salary, department, and

[5] The DB2 query plan for this simulated `STOP AFTER` version of Query 1 is a table scan, with the `Emp.salary` predicate being applied, followed by a sort to order the query result; this sort is small, occurring in memory, for small N. The actual STOP-AFTER query plan would be a table scan followed by a Sort-Stop operation; the Sort-Stop would be an in-memory operation for small N, and would require external sorting for larger N. It should be clear that the overall cost for these two execution plans is essentially the same, which is what makes this a reasonable simulation.

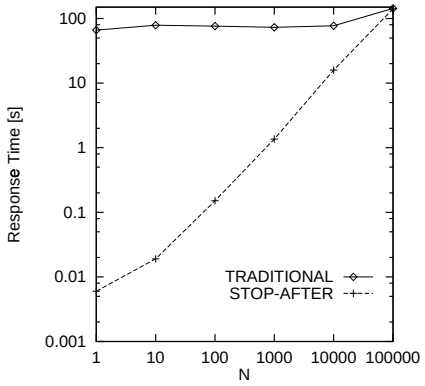


Figure 7: Resp. Time (log), Q2
Clustered Index on Emp.salary.

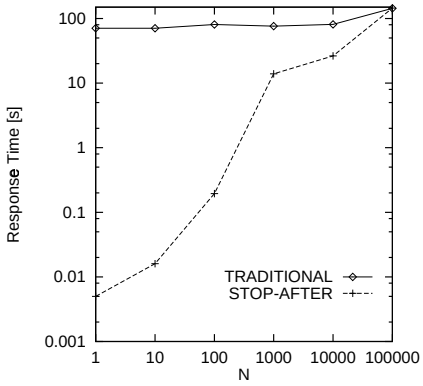


Figure 8: Resp. Time (log), Q2
Unclustered Index on Emp.salary.

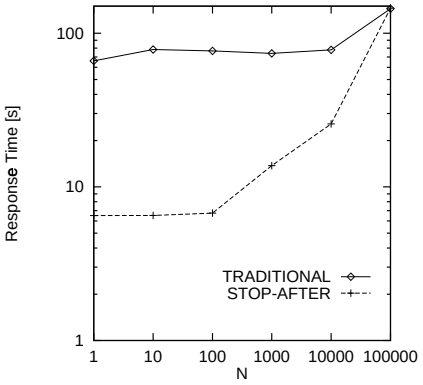


Figure 9: Resp. Time (log), Q2
No Index on Emp.salary.

Trad	Cons	*precise*	*underestimate*		*overestimate*	
		Ag(1)	Ag(1/5)	Ag(1/10)	Ag(5)	Ag(10)
128.3	63.9	6.4	33.2	63.1	6.7	18.5

Table 1: Resp. Time [s], Q3; No Idx on Emp.salary; $N = 100$

travel expenses of the N most highly paid employees.

In the previous tests, the STOP-AFTER results show what both the Conservative and Aggressive approach would produce, as the join predicate used there was non-reductive. We must distinguish the approaches here, however, as the `TEA` join predicate is (potentially) reductive and is thus a barrier to deep Stop push-down in the Conservative approach. As shown in Table 1, which contains the cost of Traditional, Conservative, and Aggressive plans for Query 3 with $N = 100$ and no index on `Emp.salary`, the difference between the Conservative and Aggressive approaches can be quite large. If the optimizer's cardinality estimates are precise (the Ag(1) column in Table 1), Aggressive outperforms Conservative by a factor of 10 in this experiment. Here, the Conservative plan involved a full sort-merge join for `Emp` $\bowtie$ `TEA`, followed by a Sort-Stop and a nested-loop index join to find the matching `Dept` names.[6] The Aggressive plan has a Sort-Stop operator just above the `Emp` scan, and uses the nested-loop index join method for both joins; it therefore avoids sorting the whole `Emp` table. The Traditional plan is significantly worse than both Stop-enhanced approaches; it uses two sort-merge joins and thus sorts the whole `Emp` table three times (twice for the joins and once to produce the final query result in the right order).

Table 1 also shows the cost of Aggressive plans if the cardinality estimates are too low or too high. If the optimizer underestimates the "precise" stopping cardinality by a factor of x (the Ag(1/x) numbers), then x restarts are required to produce the full query result. In this case, restarts are very expensive, as each one requires re-scanning the `Emp` table. As a result, the Ag(1/5) plan is about 5 times, and Ag(1/10) about 10 times, as expensive as the Ag(1) plan. Both Ag(1/5) and Ag(1/10) are still better than, or at least as good as, the Conservative and Traditional plans for $N = 100$. However, for large N (not shown in Table 1), when the potential benefits of Aggressive Stop push-down decrease, the cost of Ag(1/5) and Ag(1/10) can exceed those of Conservative and even Traditional plans by up to a factor of two in the extreme case ($N = 100,000$).

If the optimizer overestimates the "precise" stopping cardinality (the Ag(x) numbers), an Aggressive plan carries more `Emp` tuples than necessary until the final Stop; this requires no restarts, but means that too much work is done in the Join and Stop operators. In this experiment, the extra work is fairly cheap, so the performance degradation is moderate and the Ag(5) and Ag(10) plans outperform the Traditional and Conservative plans. In general, even for large N, overestimated Aggressive plans never become worse than Conservative plans—they have, at worst, the same performance. This does not make it wise to always overestimate the stopping cardinality of Aggressive Stops, however. We also conducted experiments with Query 3 and an index on `Emp.salary`. Restarts were cheap in this case (since the whole `Emp` table need not be re-scanned), while extra work was relatively expensive if the index was unclustered, so underestimation outperformed overestimation.

6 Related Work

Many commercial relational database systems allow applications to pass a hint to the optimizer to indicate that they would like the first few tuples of the query result to be produced quickly. For example, Oracle 7 has an optimization option called `FIRST ROWS` [A+92], and IBM's DB2 system offers an `OPTIMIZE FOR` N `ROWS` clause [IBM95]. The details of how (or how well) these features are implemented has not been published, but according to their reference manuals, they heuristically bias the query optimizer to more heavily favor pipelined execution plans. We experimented with DB2's `OPTIMIZE FOR` N `ROWS` construct. Its plans were as efficient as our Stop plans for single-table queries when an index existed on the query's `ORDER BY` column; when no such index existed (so no fully pipelined plan existed), or when the query was complex (making fully pipelined plans inefficient), our Stop plans performed much better.

[6] An alternative Conservative plan with slightly lower cost would completely sort the `Emp` table and then do two nested loop index joins; we could not model this plan with our DB2 simulation approach.

Other SQL extensions related to our STOP AFTER clause have recently been proposed for decision support and multimedia queries. [KS95] proposes specifying cardinality limits by having "*rank*$(\ldots) \leq N$" predicates in the WHERE clause of the query. Conceptually, the *rank* function and the corresponding query predicates are evaluated after all other predicates, joins, and aggregations have been computed; the paper does not discuss how cardinality limits can be exploited earlier to reduce join and sorting costs. [CG96] proposes an ORDER BY [N] clause that is similar to (though less powerful than) our STOP AFTER clause. The paper does discuss the implications of the new clause on query processing, but focuses on dealing with the limited query interfaces of multimedia data sources (e.g., joins are not addressed) and their execution cost features (e.g., expensive predicates and ranking expressions); as a result, their optimization framework and query processing techniques are very different than ours.

Related work on relational query processing has concentrated on developing pipelined join methods or cost models that can predict the cost to obtain the first row of a query's result set (see [G+92, WA91, BM96], among others). To our knowledge, none has proposed a Stop operator or studied how to achieve deep Stop pushdown for STOP AFTER queries.

7 Conclusions

We have examined the opportunities and query processing issues raised by adding a STOP AFTER clause to SQL's SELECT statement. We discussed how STOP AFTER support can be added to an existing DBMS, encapsulating the details of STOP AFTER queries by adding Stop operators to its query execution system. Doing so makes it unnecessary to change the rest of the system's query execution engine; its existing operators for scan, join, and so on are unaffected. We discussed how to implement the Stop operator physically, describing Scan-Stop and Sort-Stop realizations of this operator and their costs. Finally, we proposed and empirically evaluated two policies for STOP AFTER plan generation, Conservative and Aggressive, and explained how they can be realized in a rule-based query optimizer. The Conservative approach only permits Stop operators to be positioned at plan points where they are sure not to eliminate tuples that should ultimately participate in the query result; in contrast, the Aggressive approach is more adventurous, using required cardinality estimation to aggressively limit intermediate result sizes.

We used a synthetic database and query translation scheme to "simulate" our two approaches on DB2, comparing them to each other and to the traditional "just use a cursor" approach. We saw that, as anticipated, orders-of-magnitude performance gains can be achieved. The benefits of specialized STOP AFTER handling were seen to be particularly pronounced when an index is available on the query's ORDER BY column(s). The Conservative approach always provided superior performance with respect to the traditional approach; the Aggressive approach was shown to offer further improvements at the price of introducing some sensitivity to cardinality estimation errors.

In terms of future work, we plan to implement STOP AFTER in the Garlic system [C+95] at IBM Almaden. On the optimizer side, we also plan to explore techniques for handling complex STOP AFTER clauses (with subqueries that share common subexpressions with the query body) and for pushing Stop operators down into queries with multiple query blocks. On the runtime system side, we plan to design a more efficient external Sort-Stop operator and to explore join methods to handle STOP AFTER queries with multi-table ORDER BY expressions (drawing on Fagin's A_0 work [Fag96] in the join case).

Acknowledgments We would like to thank Eugene Shekita for reading through the paper and making several very helpful suggestions. Manish Arya helped with the DB2 installation.

References

[A+92] E. Armstrong, et al. *ORACLE7 server – application developer's guide*, Oracle Corporation, 1992.

[BM96] R. Bayardo and D. Miranker. Processing queries for the first few answers, *Proc. 3rd CIKM Conf.*, Rockville, MD, 1996.

[C+95] M. Carey et al. Towards heterogeneous multimedia information systems, *Proc. IEEE RIDE Workshop*, Taipei, Taiwan, 1995.

[CG96] S. Chaudhuri and L. Gravano. Optimizing queries over mulitmedia repositories, *Proc. ACM SIGMOD Conf.*, Montreal, Canada, 1996.

[CS94] S. Chaudhuri and K. Shim. Including group-by in query optimization, *Proc. 20th VLDB Conf.*, Santiago, Chile, 1994.

[Fag96] R. Fagin. Combining fuzzy information from multiple systems, *Proc. ACM PODS Conf.*, Montreal, Canada, 1996.

[GD87] G. Graefe and D. J. DeWitt. The EXODUS optimizer generator, *Proc. ACM SIGMOD Conf.*, San Francisco, CA, 1987.

[G+92] S. Ganguly, et al. Query optimization for parallel execution, *Proc. ACM SIGMOD Conf.*, San Diego, CA, 1992.

[IBM95] IBM Corporation. *DB2 application programming guide for common servers (version 2)*, 1995.

[Knu73] D. E. Knuth. *The Art of Computer Programming/Sorting and Searching*, volume 3, Addison-Wesley, 1973.

[KS95] R. Kimball and K. Strehlo. Why decision support fails and how to fix it, *SIGMOD Record*, 24(3):92–97, 1995.

[Loh88] G. Lohman. Grammar-like functional rules for representing query optimization alternatives, *Proc. ACM SIGMOD Conf.*, Chicago, IL, 1988.

[MS93] J. Melton and A. Simon. *Understanding the new SQL: a complete Guide*, Morgan-Kaufmann Publishers, Inc., 1993.

[S+79] P. Selinger, et al. Access path selection in a relational database management system, *Proc. ACM SIGMOD Conf.*, Boston, MA, 1979.

[S+96] D. Simmen, et al. Fundamental techniques for order optimization, *Proc. ACM SIGMOD Conf.*, Montreal, Canada, 1996.

[Sto96] M. Stonebraker. *Object-Relational DBMSs: The Next Great Wave*, Morgan Kaufmann Publishers, Inc., 1996.

[WA91] A. Wilshut and P. Apers. Dataflow query execution in a parallel main memory, *Proc. 1st PDIS Conf.*, Miami, FL, 1991.