Processing *Top N* and *Bottom N* Queries

Michael J. Carey
IBM Almaden Research Center
San Jose, CA 95120

Donald Kossmann
University of Passau
94030 Passau, Germany

1 Introduction

In certain application areas, such as those related to decision support or multimedia data, users wish to ask so-called *top N* and *bottom N* queries; these are queries that request a certain number of answers (N) having the highest or lowest values for some attribute, expression, or function. For example, rather than finding all publications on a certain topic, a researcher may want to retrieve the ten most heavily referenced papers on the topic at hand. A politician planning his or her next campaign might be interested in discovering the average salary of the wealthiest ten percent of the voters in a given district. Parents of a young child might want to find five mystery books that least well match the terms "crime" and "murder." These examples illustrate a variety of situations in which *top N* and *bottom N* queries are meaningful. In addition, they demonstrate the fact that such queries can involve standard relational data as well as text or other multimedia data.

To date, the SQL standard does not include statements that allow users to pose such *top N* and *bottom N* queries. There have, however, been several proposals in the literature (e.g., [KS95, CG96, CK97]), and database system vendors are beginning to extend their SQL dialects and query interfaces in order to support such queries. Given the obvious need and this growing interest, this paper addresses the question of how *top N* and *bottom N* queries can be processed efficiently; moreover, we address the question of how such support can be provided as a natural extension of existing relational query processing architectures. In a nutshell, our goal is to evaluate such queries with as little *wasted work* as possible. That is, if a query asks for the 10 most popular publications, we want to avoid work to process, say, the 11th, 12th, or 13th most popular publications.

We will begin by presenting a series of situations in which a traditional DBMS, i.e., one without integrated support for *top N* and *bottom N* queries, would end up wasting work. We then show how such a traditional DBMS could be extended – with relatively little effort, in fact – in order to avoid such wasted work and thereby achieve orders-of-magnitude improvements in many cases. Our goal here is to drive home the point that database systems must be extended in order to process *top N* and *bottom N* queries efficiently and to briefly touch upon each of the required extensions; a detailed description of our approach, as well as a performance evaluation, can be found in [CK97]. We will focus here on SQL and relational databases, using relational queries as examples and citing some measurements from a relational DBMS platform to illustrate the important performance gains that can be achieved; we note, however, that most of the effects and techniques discussed in this paper are applicable to any kind of database system.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 Forms of Wasted Work

In this section, we discuss five example queries that illustrate ways in which a traditional DBMS, which does not have built-in support for *top N* and *bottom N* queries, will end up wasting work when faced with applications that require this type of query support. These queries illustrate different forms of wasted work that occur in a traditional DBMS. We also explain how an enhanced system would be able to avoid the wasted work in each case. In addition, we cite measurements that we obtained by "simulating" our proposed system enhancements using IBM's DB2 for Common Servers DBMS product; the proposed enhancements were simulated by adding predicates to the test queries such that DB2 chose the same or equivalent plans that would be used by an enhanced system. The measurements clearly show that orders-of-magnitude performance improvements can be achieved by enhancing a DBMS with explicit support for *top N* and *bottom N* queries.[1]

To specify the example queries, we will use the syntax that we proposed in [CK97]; that is, the queries will be standard SQL queries with an additional (non-standard) `STOP AFTER` clause to specify the number of requested answers (i.e., N). In the following, we will first define the test database and then discuss the five examples.

2.1 The Test Database

To demonstrate the different forms of wasted work, we will use a simple database that contains information about a company's employees, departments, and employees' travel expense accounts (TEAs). Specifically, the database will consist of the following three tables:

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

Here, the underlined columns represent the primary keys of the tables, and the italicized columns represent foreign keys. In addition to key and referential integrity constraints, we assume that the database has constraints to enforce the fact that every employee works in some department and that every employee has a travel expense account; i.e., we assume that there are `NOT NULL` constraints in addition to foreign key constraints on the columns `Emp.works_in` and `Emp.teaNo`.

For our performance experiments, we generated synthetic `Emp`, `Dept`, and `TEA` tuples. For the measurements cited here, the `Emp`, `Dept`, and `TEA` tables had 10,000 tuples of 100 bytes each; i.e., all three of these tables contained 10 MB of data. Furthermore, we generated (clustered) primary indexes on `Emp.empId`, `Dept.dno`, and `TEA.accountNo`, and unless the text says otherwise, there was a secondary (and unclustered) index available on `Emp.salary` in each experiment. All of the indexes are B^+ trees, so they can be used in order to evaluate range and join predicates as well as reading data out of the corresponding tables in indexed-column order.

2.2 Example 1: Shipping Query Results

Our first example demonstrates the simplest form of wasted work. This form arises when the run-time system of the DBMS does not know how many tuples are desired by the application; in the absence of SQL support for *top N* or *bottom N* queries, the desired result-limiting effect will end up being implemented, e.g., simply by closing a cursor early at the application level. Specifically, this example addresses the impact of a feature called *row blocking*, which is implemented in most commercial database systems in order to reduce the overhead of shipping query results from the DBMS back to an

[1] Details of the experiments that produced the results that we cite here can be found in [CK97].

application process. Using row blocking, the DBMS allocates a certain number of buffers for a given query. It fills these buffers with tuples of the query result and then ships the whole *block* of result tuples to the application process when the buffers become full. The application process reads the query results in a tuple-at-a-time fashion using, e.g., the SQL cursor interface. Once all the tuples of the first block have been consumed by the application, the DBMS produces more query results and ships the next block of result tuples to the application process.

Row blocking is a simple yet important mechanism for efficient client-server query processing. The alternative, returning query results one tuple at a time, can become prohibitively expensive when a query result is large. For *top N* and *bottom N* queries, however, row blocking can be the cause of wasted work if the DBMS does not know the value of N and must therefore assume that the application will consume all of the tuples that satisfy a given query predicate. We will demonstrate this effect with the following example query, which asks for the first one hundred tuples of the `Emp` table (i.e., for the 100 `Emps` with the smallest *id* values):

```
SELECT     *
FROM       Emp e
ORDER BY   e.id
STOP AFTER 100
```

If the DBMS's result buffers have room for 500 tuples, a traditional DBMS would respond to this query by producing the first 500 `Emps` (using the primary index on `Emp.id`) and shipping them to the application process. In contrast, an enhanced DBMS would know that the application only wants 100 tuples, and it would therefore only produce 100 `Emp` tuples and send them as a block to the application process. In our experiments, the enhanced approach outperformed the traditional approach by a factor of four in this case:

Traditional	Enhanced
0.285s	0.075s

Two forms of wasted work are causing this performance difference – the traditional DBMS fetches too many tuples out of the `Emp` table, and it ships too much data back to the application process, both of which are unnecessary costs from the application's perspective. Obviously, the advantages of the enhanced approach become even more pronounced in this example when fewer tuples are requested by the application; the same is true of the other examples presented in this section as well.

2.3 Example 2: Access Path Selection

The second example demonstrates a basic reason why the query optimizer of a database system needs to know the value of N when optimizing a *top N* or *bottom N* query. The query in this example asks for the 100 best paid `Emps`:

```
SELECT     *
FROM       Emp e
ORDER BY   e.salary DESC
STOP AFTER 100
```

Assuming that there is a non-clustered index on `Emp.salary`, the best way to execute this query would be to use this index to find the 100 best paid `Emps` and then fetch them from the `Emp` table. Unfortunately, without the knowledge that only 100 `Emps` are being requested (i.e., assuming that all `Emps` must be returned in `salary` order), a traditional query optimizer would choose a different plan: it would choose a plan with a table scan followed by a sort operator, as the use of an unclustered index

would have an extremely high estimated cost for disk seeks if the whole Emp table with 10,000 tuples is assumed to be read. Our experiments showed that, for this example, a stop-enhanced query optimizer would outperform a traditional optimizer by over two orders of magnitude:

Traditional	Enhanced
15.633s	0.076s

2.4 Example 3: Cost of Sorting

The third example shows that an enhanced database system requires new, specialized query operators in order to process certain kinds of *top N* and *bottom N* queries efficiently. Let us reconsider the query from the previous subsection, which asked for the 100 best paid Emps. Let us now assume that there is no index available on Emp.salary. A traditional system would again execute this query with a conventional sort operator. However, a better approach to find and sort only the 100 best paid Emps is to build a priority heap in main memory [Knu73]. The first 100 Emps would be inserted into that priority heap and, after that, every Emp tuple would be inspected to see if it has higher salary than the *bottom* Emp in the heap. If so, the tuple would be inserted into the heap, thereby replacing the current bottom Emp of the heap; if not, the tuple would simply be discarded because it cannot possibly be part of the requested query result. Our experiments showed that this stop-enhanced approach to executing this example query outperforms the traditional approach by a factor of three because it avoids the wasted work of sorting (many) tuples which are not part of the desired query result:

Traditional	Enhanced
15.633s	5.775s

2.5 Example 4: Pipelined Join Methods

The fourth example again demonstrates a potential improvement that can be achieved by enhancing the query optimizer of a traditional DBMS. This example shows that many *top N* and *bottom N* queries are best executed using a pipelined query plan; specifically, this example shows that a nested-loop index join can be a very efficient method to evaluate a multi-table *top N* and *bottom N* query. The following example query requests the employee and department information for the 100 best paid Emps:

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

If both tables are large, the optimizer of a traditional system, which is oblivious to the desired final result cardinality, would likely choose a sort-merge or hybrid-hash join in order to evaluate this query. In contrast, an enhanced system would most likely generate a plan in which the Emp tuples are produced in order (using the Emp.salary index if there is one, or a sort-based operator if not) and then join one Emp tuple with the Dept table at a time using the index nested-loops method until 100 Emp tuples have qualified. Our experiments show that such an enhanced system would outperform a traditional system by two orders of magnitude in this case if there is an index available on Emp.salary:

Traditional	Enhanced
80.716s	0.195s

2.6 Example 5: Join Queries With No Good Pipelined Plan

For some *top N* and *bottom N* queries, no good pipelined query plan exists. Examples include queries with large N, where an index nested-loop join becomes too expensive for producing all N results. Another example is a multi-way join query where not all joins should be carried out in a pipelined fashion, e.g., because of excessive main-memory consumption, or because the query optimizer chooses a join order with a bushy query plan. In such situations, it is important to reduce the cardinality of intermediate results as early as possible in order to reduce the cost of subsequent expensive (e.g., join) operators. This is illustrated by the following example, which requests the department and travel expense information for the 100 highest paid employees:

```
SELECT     *
FROM       Emp e, Dept d, TEA t
WHERE      e.works_in = d.id AND t.teaNo = t.accountNo
ORDER BY   e.salary DESC
STOP AFTER 100
```

Given the facts (implied by the database's integrity constraints) that every `Emp` works in a department and has a travel account, an efficient way to execute this query is to identify the 100 best paid `Emps` first and then carry out the joins in order to find out the department and travel information for only these 100 tuples. A traditional system, however, would most probably carry out the joins first, using the whole `Emp` table, identifying the 100 best paid `Emps` with their `Dept` and `TEA` information only at the end (i.e., up in the application program). As a result, our experiments showed that a traditional system would be outperformed by an order of magnitude by a stop-enhanced system in this example:

Traditional	Enhanced
128.307s	6.381s

3 Extending a DBMS

In this section, we will describe how a traditional relational DBMS can be extended to provide good performance for all five of the examples from the previous section. Our approach is to use existing components as much as possible to process *top N* and *bottom N* queries, only adding new stop-specific code when absolutely necessary. As a result, the required changes are moderate, making it possible to implement and integrate the changes with a modest amount of effort; the handling of regular queries (i.e., non-top/bottom queries) is unchanged. After describing our approach, we will briefly discuss how some degree of support for *top N* and *bottom N* queries has recently been integrated into certain commercial database systems.

3.1 Our Approach

In the following, we list and briefly sketch out the changes that we propose in order to extend a traditional (relational) DBMS to handle *top N* and *bottom N* queries. A much more detailed description of these extensions and their impact can be found in [CK97].

SQL and Parser As mentioned earlier, we propose extending the syntax of SQL to include a `STOP AFTER N` clause as an optional suffix to SQL's `SELECT` statement; this allows users to express *top N* and *bottom N* queries. Just as any `SELECT` statement can be used as part of the definition of a subquery or a view, we allow a `STOP AFTER` clause (and a corresponding `ORDER BY` clause) to be used in the definition of subqueries and/or views. Furthermore, we will allow for N, the number of answers requested by

the application, to be provided as an expression (including support for complex expressions such as a scalar subquery). The addition of the `STOP AFTER` clause is sufficient to give the DBMS the cardinality information needed to avoid wasted work due to row blocking (see the first example of Section 2), as the DBMS then has a strict upper limit on the number of result tuples that can result from the query.

Stop Operators The second extension we propose is the addition of a *Stop* operator to the repertoire of the DBMS's query execution engine. The Stop operator produces the top (or bottom) N tuples of its input stream, taking the stopping cardinality N plus a sort expression as parameters. The Stop operator serves to encapsulate the function of the `STOP AFTER` clause; the query execution plan for a *top* N or *bottom* N query will contain at least one Stop operator. In addition, given a Stop operator, the implementation of the other (existing) operators, such as join operators, sort, group-by, and scans, does not need to be changed at all.

Like many query operators, such as join, sort, and group-by, the Stop operator can be implemented in any of several different ways. If the input of the Stop operator is already ordered according to its sorting expression, then the Stop operator will simply return the first N tuples of its input and then signals "end-of-stream." If the input is not ordered, and N is relatively small, the Stop operator can be implemented using a priority heap as described in Section 2.4. Finally, if the input is not ordered, and N is very large, the Stop operator can be implemented using a conventional external sort algorithm that discards the remaining data after identifying the first N results.

New Optimization Rules and Modified Pruning for Stop Operator Placement To correctly enumerate plans with Stop operators, the query optimizer must be extended. This extension is quite easy to implement in modern optimizers, as modern optimizers are rule-based; the enumeration of plans with Stop operators can be arranged by adding new rules related to Stop operators to the optimizer [GD87, Loh88]. Note that it is important to enumerate all possible plans with Stop operators during cost-based query optimization because Stop operators influence the cost of other operators and, as a result, are likely to impact other cost-based decisions carried out by the optimizer such as join ordering, choice of join methods, and access path selection.

In addition to defining new rules to enumerate alternative plans with Stop operators, the pruning condition of a query optimizer that is based on dynamic programming needs to be changed. It is, for example, possible for a subplan with a Stop operator and a higher cost than a corresponding subplan without a Stop operator to end up being a building block for the best overall plan for the whole query. This situation can arise because Stop operators have a non-negligible cost, and the quality of a subplan with a Stop operator can only be seen for sure upon considering operators that come later in the query plan; this is because such operators benefit from the fact that the Stop operator yields a smaller intermediate query result. The optimizer therefore cannot safely prune a plan with one or more Stop operators in favor of another (otherwise equivalent) query plan without a Stop operator.

Stop Operator Placement New rules and a modified pruning condition will ensure that all possible plans with Stop operators are enumerated and that no query plans with Stop operators are pruned prematurely. Of course, we also need to decide *where* Stop operators are to be placed in a query plan. As shown in the fifth example of Section 2, Stop operators should be located as early as possible in a query plan so that the cost of subsequent operators will be reduced. On the other hand, Stop operators should ideally only be placed at *safe* points, i.e., at points in a query plan where their presence can never cause tuples to be discarded that may end up being needed in order to generate the requested N tuples of the query result. Thus, the goal is to find the earliest *safe* place for a Stop operator in a query plan.

Safe places for Stop operators in a query plan can be found by inspecting the database's integrity constraints together with the given query's predicates (i.e., the query's `WHERE` clause). For the `Emp-Dept-TEA` query of Section 2.6, for example, placing a Stop operator below the joins with `Emp-Dept` and `Emp-TEA` joins was safe because the referential and `NOT NULL` integrity constraints on `Emp.works_in` and `Emp.teaNo` guarantee that every `Emp` tuple will satisfy both join predicates; consequently, they ensure that (at least) 100 result tuples will be produced from joins involving the 100 best paid `Emps`. Now consider a different example, where the user asks for the `Dept` and `TEA` information for the 100 best paid `Emps` that work in a "research" department. In this case, a Stop operator could safely be placed below the `Emp-TEA` join, as before. However, it could not safely be placed below the `Emp-Dept` join unless there is an integrity constraint that guarantees that every department is a research department (because, in general, not all `Emp` tuples will survive a join with only the research `Dept` tuples).

In [CK97], we also studied *unsafe* (a.k.a. *aggressive*) Stop operator placement techniques; these techniques can be applied even if there are no appropriate integrity constraints. They would, for instance, allow the placement of a Stop operator below the `Emp-Dept` join in the "find the 100 best paid `Emps` that work in a research department" example discussed above. In this case, the optimizer will estimate the number of tuples that should be filtered out by the (unsafe) Stop operator by considering cardinality and selectivity estimates and working backwards from the desired overall result cardinality. For example, if every other `Emp` works in a research department, the unsafe Stop operator will be instructed to stop after returning the 200 best paid `Emps`; these 200 tuples would be expected to include the requested 100 result tuples. Of course, cardinality estimates can easily be too high or too low, so that precautions must be taken to deal with such imprecise cardinality estimates. To deal with cardinality estimates that are too high, a final Stop operator is required at the top of the plan in order to make sure that no more than 100 query results are produced; to deal with overly low cardinality estimates, the plan must somehow be *restartable* at run-time in order to produce the missing tuples which were discarded by the unsafe Stop operator.

In [CK97], we studied the tradeoffs associated with an approach that allows *unsafe* Stop operators. We saw that in some cases, it is indeed better to have low-level *unsafe* Stop operators even when the cardinality estimates are imprecise. However, in other cases, we observed that excessive restarts due to overly low estimates hurt performance dramatically; in extreme cases, the performance of the enhanced system with *unsafe* Stop operators even dropped below the performance of a traditional system with no support for *top N* and *bottom N* queries. In contrast, an enhanced system with only safe Stop operator placement can never be outperformed by a traditional system.

3.2 Support in Current Database Systems

As stated in the introduction, several database vendors have recently added features that provide some degree of support for top and bottom queries. Unfortunately, none has published information how they have implemented those features. Thus, here we will discuss only the current version of DB2 for Common Servers (Version 2.1.1), as this is the only system that we were able to learn about and experiment with for the examples presented in Section 2. [2]

DB2 allows users to specify that they want the first N answers of a query quickly via an `OPTIMIZE FOR N ROWS` clause that can be given as an optional suffix for `SELECT` queries. The presence of an `OPTIMIZE FOR N ROWS` clause influences query processing in DB2 in two ways. First, it passes the value of N to the run-time system of DB2 so that DB2 can adjust its row blocking; as a result, DB2 can do well on the first example of Section 2 if the query is given as an `OPTIMIZE FOR 100 ROWS` query.

[2] From comments in the manuals, it seems that Oracle 7 has followed an approach similar to that of DB2 [ABF+92].

Second, the presence of the `OPTIMIZE FOR N ROWS` clause makes the DB2 optimizer favor pipelined plans; this favoritism would allow DB2 to also perform well in the second and fourth examples of Section 2 [Cor95]. It is important to note, however, that DB2's `OPTIMIZE FOR N ROWS` clause is only a *hint* that instructs DB2 to produce the first N answers quickly; users are allowed to consume more than N tuples (at potentially lower performance) even when they give such a hint, whereas our `STOP AFTER` clause instructs a DBMS to produce exactly N (and never more) answers. DB2 thus never discards answer tuples, preventing it from exploiting some of the significant cost savings that are available through the use of specialized Stop operators (especially in non-pipelined query plans, such as those needed in the third and fifth example of Section 2). DB2 has not (yet) integrated any of the techniques of our approach described above.

4 Conclusion

We have shown various ways in which traditional database systems are susceptible to wasted work when evaluating *top N* and *bottom N* queries. We have seen that, in many cases, orders-of-magnitude improvements can be achieved with moderate extensions to the parser, query engine, and optimizer of an existing database system.

Most of the discussion of this paper has been concerned with *top N* and *bottom N* queries where the requested result cardinality (N) was specified in the query as part of a basic `STOP AFTER N` clause. Another important class of queries are *percent* queries, which ask for, e.g., the top P percent of the answer set rather than the top N answers. Percent queries offer even more opportunities to avoid wasted work, though they also require additional work in order to determine what $P\%$ of the answer really is (cardinality-wise). We plan to study good strategies for percent query processing as future work; in addition, we plan to study specialized algorithms for sorts and joins in the context of any kind of top or bottom query, and and we plan to investigate new techniques for processing top and bottom queries in the context of subqueries and views.

References

[ABF+92] E. Armstrong, S. Bobrowski, J. Frazzini, B. Linden, and M. Pratt. *ORACLE7 Server – Application Developer's Guide.* Oracle Corporation, Redwood Shores, USA, 1992.

[CG96] S. Chaudhuri and L. Gravano. Optimizing queries over mulitmedia repositories. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 91–102, Montreal, Canada, June 1996.

[Cha96] D. Chamberlin. *Using the New DB2: IBM's Object-Relational Database System.* Morgan-Kaufmann Publishers, San Mateo, USA, 1996.

[CK97] M. Carey and D. Kossmann. On saying "enough already!" in SQL. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 219–230, Tucson, USA, May 1997.

[Cor95] IBM Corporation. DB2 application programming guide for common servers. Manual, 1995.

[GD87] G. Graefe and D. DeWitt. The EXODUS optimizer generator. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 160–172, San Francisco, USA, May 1987.

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

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

[Loh88] G. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 18–27, Chicago, USA, May 1988.