Reducing the Braking Distance of an SQL Query Engine

Michael J. Carey
IBM Almaden Research Center
San Jose, CA 95120
carey@almaden.ibm.com

Donald Kossmann
University of Passau
94030 Passau, Germany
kossmann@db.fmi.uni-passau.de

Abstract

In a recent paper, we proposed adding a `STOP AFTER` *clause to SQL to permit the cardinality of a query result to be explicitly limited by query writers and query tools. We demonstrated the usefulness of having this clause, showed how to extend a traditional cost-based query optimizer to accommodate it, and demonstrated via DB2-based simulations that large performance gains are possible when* `STOP AFTER` *queries are explicitly supported by the database engine. In this paper, we present several new strategies for efficiently processing* `STOP AFTER` *queries. These strategies, based largely on the use of range partitioning techniques, offer significant additional savings for handling* `STOP AFTER` *queries that yield sizeable result sets. We describe classes of queries where such savings would indeed arise and present experimental measurements that show the benefits and tradeoffs associated with the new processing strategies.*

1 Introduction

In decision support applications, it is not uncommon to wish to pose a query and then to examine and process at most some number (*N*) of the result tuples. In most database systems, until recently, applications could only do this by using a cursor, i.e., by submitting the entire query and fetching only the first *N* results. Obviously, this can be very inefficient, leading to a significant amount of wasted query processing. In a recent paper [CK97], we proposed adding a `STOP AFTER` clause to SQL to enable query writers to limit the size of a query's result set to a specified number of tuples; related SQL extensions have been proposed in [KS95, CG96]. The `STOP AFTER` clause essentially provides a declarative way for a user to say ”enough already!” in the context of an SQL query, enabling the system to avoid computing unwanted results in many cases. In our previous work we showed the usefulness of the new clause, discussed how a cost-based query optimizer can be extended to exploit it, and used DB2-based simulations to demonstrate the large performance gains that are possible when `STOP AFTER` query support is explicitly added to the database engine.

In this paper, we build upon our previous work by presenting several new strategies for efficiently processing `STOP AFTER` queries. Although we discussed `STOP AFTER` query processing in general in [CK97], the major focus of our initial attention was on optimizing queries where *N* is relatively small (e.g., “top ten” queries). An example of a typical query that our previously proposed processing schemes will handle well is:

```
SELECT e.name, e.salary
FROM Emp e
WHERE e.age > 50
ORDER BY e.salary DESC
STOP AFTER 10;
```

This query asks for the names and salaries of the ten most highly-paid older employees in the company. Our previous schemes will also work well for primary-key/foreign-key join queries such as:

```
SELECT e.name, e.salary, d.name
FROM Emp e, Dept d
WHERE e.age > 50
AND e.works_in = d.dno
ORDER BY e.salary DESC
STOP AFTER 10;
```

This query asks for the employees' department names as well as their names and salaries. For queries such as these, it is possible for the query processor to manage its sorted, cardinality-reduced intermediate results using a main memory heap structure, thereby avoiding large volumes of wasted sorting I/O as compared to processing the query without a `STOP AFTER` clause and then discarding the unwanted employee information.

In cases where the stopping cardinality *N* is large, our original approaches would each end up sorting and then discarding a significant amount of data—albeit early (i.e., before the join in the example above), which still leads to a significant savings compared to the naive approach. The strategies presented in this paper seek to avoid this wasted effort as well. Our new strategies are based upon borrowing ideas from existing query processing techniques

Proceedings of the 24th VLDB Conference
New York, USA, 1998

such as range partitioning (commonly used in parallel sorting and parallel join computations), RID-list processing (commonly used in text processing and set query processing), and semi-joins (commonly used in distributed environments to reduce join processing costs). As we will show, adapting these techniques for use in STOP AFTER query processing can provide significant additional savings for certain important classes of queries. We have implemented the techniques in the context of an experimental query processing system at the University of Passau, and we will demonstrate the efficacy of our techniques by presenting measurements of query plans running there.

Before proceeding, it is worth noting that proprietary SQL extensions closely related to our proposed STOP AFTER clause can be found in current products from a number of major database system vendors. In addition, most of them include some degree of optimizer support for getting the first query results back quickly (e.g., heuristically favoring pipelined query plans over otherwise cheaper, but blocking, non-pipelined plans). For example, Informix includes a FIRST_ROWS optimizer hint and a FIRST n clause for truncating an SQL query's result set. Similarly, Microsoft SQL Server provides an OPTION FAST n clause and a session-level SET ROWCOUNT n statement for these purposes. IBM's DB2 UDB system allows users to include OPTIMIZE FOR n ROWS and/or FETCH FIRST n ROWS ONLY clauses when entering an SQL query. Oracle Rdb (originally a DEC product) added a LIMIT TO n ROWS clause to SQL, while Oracle Server makes a virtual ROWNUM attribute part of its query results to support cardinality limits; including the predicate ROWNUM <= n in the WHERE clause of an SQL query tells Oracle Server to stop returning result rows after n rows have been produced. RedBrick supports a SET ROWCOUNT n command as well as an SQL extension called RANK(col) which both imposes a result order and allows processing to be stopped early; adding the clause WHEN RANK(col) < n to a query tells RedBrick to return the result rows that rank among the first n column values with respect to the indicated column. (In the event of a tie, RedBrick permits multiple result rows to have the same rank value.) Finally, several of these systems apparently pass stopping information to operations such as *Sort* so that they can optimize for the desired number of results when merging sorted runs. Unfortunately, to the best of our knowledge, there is no published information available that describes how any of these systems' SQL extensions are implemented.

The remainder of this paper is organized as follows: We present background material in Section 2, where we briefly summarize the query operators and kinds of query plans introduced in our previous work, review the basic idea of range partitioning as a query processing step, and provide an overview of the experimental environment used to produce the performance results presented in later sections of the paper. In Section 3 we introduce our new range-based techniques for processing STOP AFTER queries and present experimental results that demonstrate their benefits and highlight their associated performance issues and tradeoffs. We focus on basic top N selection queries in Section 3, while in Section 4 we explain how range techniques can be utilized for processing queries such as top N percentage selections, selections involving STOP AFTER subqueries, and joins. In Section 5 we show how RID-list and semi-join techniques can be applied to STOP AFTER queries. Finally, we present our conclusions and our plans for future work in Section 6.

2 Background

The general structure proposed for STOP AFTER queries (and subqueries) in [CK97] is as follows:

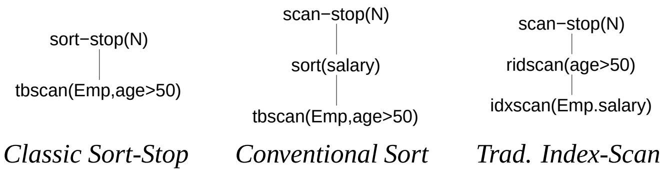
```
SELECT ...  FROM ...  WHERE ...
GROUP BY ...  HAVING ...
ORDER BY ⟨sort specification list⟩
STOP AFTER ⟨value expression⟩
```

The STOP AFTER clause's <value expression> evaluates to a scalar integer value to indicate the number of result tuples desired; it may be a constant, an arithmetic expression, or even an uncorrelated scalar subquery. The semantics of the STOP AFTER clause are straightforward to explain: Let N be the integer stopping cardinality that <value expression> evaluates to. After computing the rest of the query, the system is to return only the first N tuples of the result (in the specified ORDER BY order, if any) to the requesting user or application program. Note that this produces the same results as the cursor-based approach used by application programs today, but the presence of the STOP AFTER clause provides the query optimizer and runtime query processing system with cardinality information that can be exploited to reduce (or even eliminate, in some cases) wasted work.

2.1 STOP AFTER Query Processing

To process STOP AFTER queries, we proposed extending the database system's collection of algebraic query operators with a new logical query operator, the *Stop* operator. This operator produces the *top* or *bottom* N tuples of its input stream in a specified order and discards the remainder of the stream. Like other logical query operators (such as Join), Stop has several alternative physical operators that can implement it in the context of query plans.

We defined two physical Stop operators in [CK97]: *Scan-Stop*, for use when the Stop operator's stream of input tuples is already ordered appropriately, and *Sort-Stop*, for use when the Stop operator's input stream is not yet rank-ordered. Scan-Stop is extremely simple; it is a pipelined operator that simply requests and then passes each of the first N tuples of its input stream along to its consumer (i.e., to the operator above it in the query plan). In contrast, the Sort-Stop operator handles the case where the input stream is *not* already sorted; it must therefore consume its whole input stream in order to produce the *top* (or *bottom*) N output tuples. When N is relatively small, Sort-Stop can operate in main memory using a priority heap [Knu73]. The



Classic Sort-Stop *Conventional Sort* *Trad. Index-Scan*

Figure 1: Traditional Plans for Query 1

first N tuples of its input are inserted into the heap, and each remaining tuple is then incrementally tested against the heap's current membership bound to determine whether or not it warrants insertion into the heap of the top (or bottom) N tuples. For larger values of N, external sorting is required to compute the desired Sort-Stop results; we simply used an ordinary external Sort operator followed by a Scan-Stop operator in such cases in [CK97].

For illustration purposes, consider a slightly more general version of the first example query from the Introduction (we will call this Query 1 in the following):

Query 1:

```
SELECT      *
FROM        Emp
WHERE       age > 50
ORDER BY    salary DESC
STOP AFTER N;
```

Figure 1 depicts three of the possible execution plans that can be constructed for this query by combining one of our physical Stop operators with other, pre-existing query operators. The first plan, the *Classic Sort-Stop* plan, uses a table scan (`tbscan`) operator to find employees in the appropriate age range followed by a heap-based `sort-stop(N)` operator to limit the results to the N highest paid older employees. This plan is viable as long as N is small enough for the heap to indeed be a main memory structure. The second plan, *Conventional Sort*, instead uses an external sort on `salary` followed by a `scan-stop(N)` to obtain the desired result. This would be the preferred plan for large N in the absence of a `salary` index. Of course, plans similar to these two, but with an `Emp.age` index scan used to produce the inputs to the Stop-related operators, are possible as well. The third plan in Figure 1, the *Traditional Index-Scan* plan, would also become viable in the presence of an index on `Emp.salary`. This plan performs an index scan (in descending order) on the salary index, uses the resulting record ids (RIDs) to fetch high-salaried employees and applies the `age` predicate to them, and then uses a `scan-stop` operator to select the top N results since the index scan produces its output in the desired `salary` order. This third plan does very well if the salary index is a clustered index or N is small. If N is large and the index is unclustered, however, it would do too many random I/Os to be cost-effective, especially if the `age` predicate is highly selective (in which case many of the high-salaried employees found using the index would subsequently be eliminated).

We introduced two policies to govern the placement of Stop operators in query plans in [CK97]. One was a *Conservative* policy, which inserts Stop operators as early as possible in a query plan subject to the constraint that no tuple that might end up participating in the final N-tuple query result can be discarded by a Stop operation. We also proposed an *Aggressive* policy that seeks to introduce Stop operators in a query plan even earlier, placing a Stop operator wherever it can first provide a beneficial cardinality reduction. The Aggressive policy uses result size estimation to choose the stopping cardinality for the Stop operator; at runtime, if the stopping cardinality estimate turns out to have been too low, the query is *restarted* in order to get the missing tuples. This is accomplished by placing a `restart` operator in the query plan; this operator's job is to ensure that, above its point in the plan, all N tuples will be generated. Thus, if its input stream runs out before all N tuples are received, it will "restart" the query subplan beneath it to obtain the missing results.

2.2 Range Partitioning

Range partitioning is a well-known technique that has been applied with much success to numerous problems in the parallel database algorithm area [DG92]. One successful example is parallel sorting [DNS91b], while another is load-balanced parallel join computation [DNSS92]; yet another example is the computation of so-called band joins [DNS91a]. The basic idea of range partitioning is extremely simple—the data is divided into separately processable buckets by placing tuples with attribute values in one range into bucket #1, tuples with attribute values in the next range into bucket #2, and so on. In the case of parallel sorting, each node in a *k*-node database machine partitions its data into *k* buckets in parallel, based on the sorting attribute(s), streaming each bucket's contents to that bucket's designated receiver node while the data is being partitioned. At the end of this process, the individual buckets can be sorted in parallel with no further inter-node interaction. Figure 2 illustrates this process. Successful partitioning in this manner produces virtually linear sorting speedup, and sampling (or histogram) techniques can aid in the determination of a good set of partition boundary values [DNS91b] at relatively low cost.

In Section 3, we will propose and analyze the use of several possible partitioning-based approaches for improving the efficiency of `STOP AFTER` query processing. We will cover the details later, but the basic idea is simple—the relevant data can be range-partitioned on the query's `ORDER BY` attribute into a number of buckets. The buckets can then be processed one at a time until N results have been output; buckets that are not accessed in this process need never be sorted at all. This provides a way to implement a Stop(N) operation that scales beyond main memory sizes without requiring full sorting of the input stream. In addition, we will see that in certain contexts, additional significant savings are possible, e.g., cases involving uses of a `STOP AFTER` clause in a subquery.

2.3 Experimental Environment

As we work our way through the presentation of the proposed new approaches for executing `STOP AFTER`

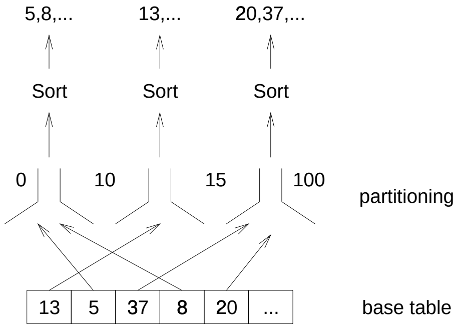


Figure 2: Sorting by Range Partitioning

queries, we will be presenting results from performance experiments that demonstrate the tradeoffs related to the approaches and that quantitatively explore the extent to which they are able to reduce the costs of STOP AFTER queries. Like Query 1 above, our test queries will be queries over a simple employee database with the following self-explanatory schema:

```
Emp(eno, works_in, age, salary, address)
Dept(dno, budget, description)
```

Our instance of this employee database is fairly small, with a 50 MB Emp table and a 10 MB Dept table. We kept the database small in order to achieve acceptable running times and because we had somewhat limited disk space available for performing our experiments. The Emp table has 500,000 tuples which are generated as follows: eno is set by counting the tuples from 1 to 500,000, while works_in, age, and salary are set randomly using a uniform distribution on their particular domains; address simply pads the tuples with “garbage” characters to ensure that each Emp tuple is 100 bytes long. The domain of works_in is, of course, the same as that of Dept.dno (described below), the domain of age is integers in the range from 10 to 60 so that about 100,000 Emps (20%) are older than 50, and the domain of salary is integers in the range of 1 to 500,000. Our test database has no correlations; as an example relevant to our experiments, a young Emp is just as likely to have a high salary as an old Emp is.

The Dept table has 100,000 tuples which are generated as follows: dno is set by counting the Dept tuples from 1 to 100,000; budget is set to 10,000 for all Depts, and description pads the Dept tuples out to 100 bytes.

In terms of indexes, our test database has clustered B^+ tree indexes on the primary key attributes of the tables (i.e., eno and dno) because clustered indexes on primary keys are relatively common. To study plans such as the Traditional Index-Scan plan of Figure 1, we also have an Emp.salary B^+ tree; naturally, this index is unclustered.

Our experiments have been performed on an experimental database system called AODB [WKHM98]. AODB is essentially a textbook relational database system that uses standard implementations for sorting, various kinds of joins, group-by operations, and so on. We extended AODB with implementations for the scan-stop, sort-stop (using 2-3 trees to organize the heap [AHU83]), and restart operators described above; we also added support for the forms of range partitioning described in the next section. We ran AODB on a Sun workstation 10 with a 33 MHz SPARC processor, 64 MB of main memory, and a 4 GB disk drive that is used to store code, the database, and temporary results of queries. The operating system is Solaris 2.6, and we used Solaris' *direct I/O* feature; this disables operating system caching in a manner similar to raw disk devices. Since our database is small, we limited the size of the database buffer pool proportionally, to 4 MB, in those cases where we do not explicitly say otherwise. Of these 4 MB, we always gave at least 100 KB to each operator that reads or writes data to disk in order to enable large block I/O operations and avoid excessive disk seeks.

3 Range-Based Braking Algorithms

We now turn our attention to the development of new techniques for processing STOP AFTER queries with less effort—i.e., techniques for reducing the “stopping distance” of an SQL query engine. The primary tool that we will be using is range partitioning. In this section, we present several algorithms that use this tool to help the engine to limit wasted work, thereby finishing sooner for STOP AFTER queries; we refer to these as “range-based braking” algorithms. We start by describing query plan components that can realize the algorithms and illustrating them using a typical example query. We then study their performance, and we close this section by explaining how to choose an appropriate number of partitions and an effective set of partition sizes.

3.1 Range-Based Braking

As mentioned earlier, the problem of extracting the top (or bottom) N elements from a large data set, where N is large as well, can be dealt with by first range-partitioning the data into a number of buckets on the query's ranking attribute(s) and then processing the resulting buckets one at a time until all N elements have been output. As an example, consider again Query 1 of Section 2.1, which selects the names and salaries of the N highest paid employees over 50 years old. Let us suppose we have a corporation with 100,000 older employees (as in our test database) and that N is 10,000. We could, for instance, partition the company's old employees into three buckets—those with salaries over $250,000 per year, those who earn between $50,000 to $250,000 annually, and those who earn less than $50,000. Suppose that we do this and find that the first (highest salary) partition ends up with 1,000 tuples, the second with 12,000 tuples, and the third with 87,000 tuples. If this is the case, we need not sort the tuples in the last partition, as the 10,000 employees in the answer set clearly lie in the first two partitions.

While the basic idea of range-based braking is simple, there are several possible variations on this theme with costs and benefits that depend on the nature of the query being processed and the data being accessed. One important option has to do with how the partitions are handled: they can either be materialized (i.e., stored as temporary tables), or they can be recomputed on demand from the input data. In addition, these two options can be combined to produce a hybrid approach that materializes some of the partitions (those that are likely to be accessed, e.g., the first two partitions above) and recomputes the rest on demand (the ones that are unlikely to be accessed).

To provide for these different options, we propose adding several new query operators to the execution engine. The first is a part-mat operator, which takes a partitioning vector as a parameter and uses it to scan its input data and write it to disk in a specified number of partitions based on the splitting values given in the partitioning vector. The second is a part-scan operator that is used to scan the resulting partitions one-by-one. The third new operator is a part-reread operator, which takes a set of predicates that describe the membership criteria for every partition (e.g., {salary > 250,000, 50,000 < salary ≤ 250,000, salary ≤ 50,000}) and materializes a partition's tuples by reading (or re-reading) its input stream from the beginning. The final new query operator is a part-hybrid operator, which materializes a specified number of its highest (or lowest) ranked partitions and computes the contents of the other partitions only on demand. We will further illustrate how each of these operators works, and discuss their performance tradeoffs, by using the example plans presented in the next subsection.

3.2 Range-Based "Top N" Query Plans

To demonstrate how the different variations of range-based braking actually work, let us turn once again to Query 1, our favorite STOP AFTER query example. Figure 3 shows three possible partitioning plans for processing Query 1 in the absence of any useful indexes. (We will discuss STOP AFTER query processing with indexes in Section 5.1.) The first plan, labeled *Materialize*, takes the approach of materializing all of the employee partitions and then sorting (only) those needed to yield N results. The execution of this plan is demand-driven and best explained by looking at what happens as result tuples are requested from the scan-stop(N) operator at the top of the plan. When the first result tuple is requested, the scan-stop(N) operator attempts to obtain and produce its first result, so it asks the restart(N) operator for a tuple, which in turn asks the sort operator underneath it for a tuple. The sort operator responds by consuming and sorting all of the tuples it can get before getting an "end-of-input" indication from the part-scan operator beneath it. The part-scan operator obtains tuples by scanning the first partition produced by its child, the part-mat operator, which materializes a full set of partitions with all of the old employees by partitioning the result of its input (coming from the employee table scan) before allowing the part-scan to proceed. When the part-scan finishes scanning the first partition, it returns an "end-of-input" signal to the sort, which sorts the partition and then incrementally passes the results for the first partition to the scan-stop(N) operator through the restart(N) operator. When the restart(N) operator receives "end-of-input," it sends a restart signal back down the tuple pipeline; when this signal is received by the part-scan operator, it responds by moving on to the next partition, and so on. The result is that the partitions created by the part-mat operator are sorted, one by one, until the scan-stop(N) at the top has produced N results. Partitions not needed to achieve that goal remain unsorted, thereby saving on sorting cost as compared to the Conventional Sort plan of Figure 1.

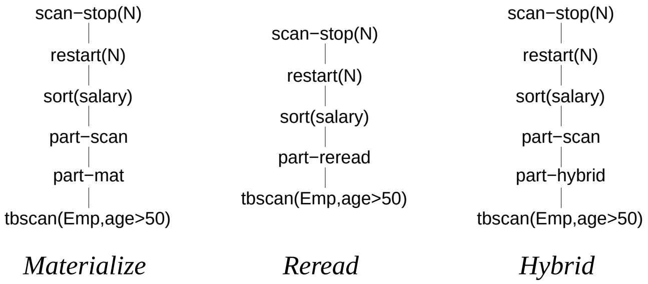


Figure 3: Range Partitioning Plans for Query 1

The second partitioning plan shown in Figure 3, the one labeled *Reread*, does not materialize its partitions as temporary files. Instead, it computes and sorts the partitions on demand by feeding a sort operator one partition at a time from a part-reread operator. Again, the plan is controlled at the top by a scan-stop(N) and a restart(N) operator. In this case, each time a partition is computed and sorted, the employee table scan will be repeated; this happens because the part-reread operator responds to a restart signal by re-initializing its input operator (i.e., the table scan in the example) or its input operator tree (for more complex query plans), which then starts over from the beginning. The execution of this query plan is otherwise similar to that described above, so hopefully its control and data flow details are now clear. The advantage of the Reread plan for our favorite query is that it saves the cost of writing and re-reading the materialized partitions; note that this can include partitions, like the large third partition in our earlier example, that are not needed at all to obtain an N-tuple result set. On the other hand, it has to re-scan the employee table for each partition that it does use, so there is a read cost associated with the write/read savings that this approach involves. The final partitioning plan which is shown in Figure 3, labeled *Hybrid*, attempts to combine the advantages of the other two plans while avoiding their disadvantages. In particular, it is structured in such a way that it materializes its first few partitions but recomputes the remaining ones.

3.3 "Top N" Select Queries

At this point, we have a collection of five query plans that all could be used to process our favorite query in the absence of indexes: the Classic Sort-Stop and Conventional Sort approaches of [CK97], shown in Figure 1,

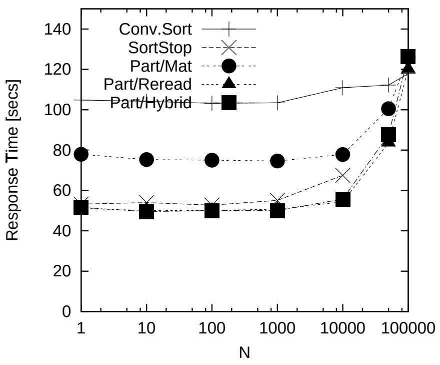


Figure 4: Resp. Time (secs), Query 1
4 MB Buffer, Perf. Part.

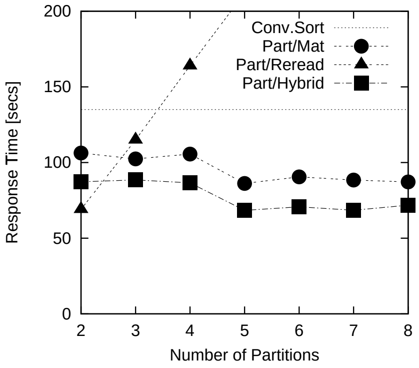


Figure 5: Query 1, Vary #Part.
600 KB Buffer, $N = 20,000$, Perf. Cut

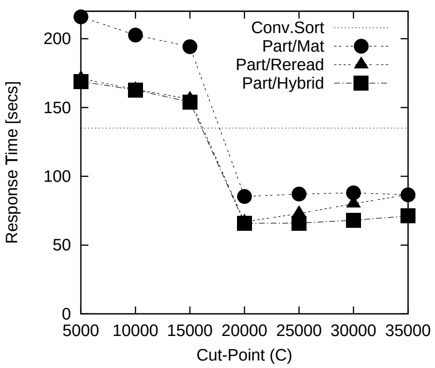


Figure 6: Query 1, Vary Cut-point
600 KB Buffer, $N = 20,000$, Perf. Part.

and the partition-based Materialize, Reread, and Hybrid approaches, which we just introduced, and which are shown in Figure 3. To investigate the quantitative tradeoffs between the five approaches, we constructed each of the query plans and conducted a series of experiments using our test employee database and the AODB system, which are both described in Section 2.3. Figure 4 shows the overall Query 1 response time results that we obtained by experimenting with stopping cardinality values ranging from 1 to 100,000 (i.e., from one up to all of the "old" employees).

The Conventional Sort plan has the worst performance throughout most of Figure 4 because it sorts all of the data before it is able to identify the top N results. The performance figures for this plan thus always include the cost of a two-phase sort of 10 MB, which takes approximately two minutes in our test environment. The cost increase for larger values of N are due to the N-dependence of the final merge phase of the sort; for $N = 1$, only the first page of each run is read, while for $N = 100,000$ all pages of all runs are read and merged. The Classic Sort-Stop plan provides much better performance than the Conventional Sort plan as long as it is applicable; its curve stops at $N = 10,000$ because its sorted heap structure no longer fits in the buffer pool beyond that point. The relative performance seen for these approaches is essentially just as predicted in [CK97].

We now turn to the three partition-based approaches in Figure 4. In this experiment, we assume that the optimizer's selectivity estimator has access to accurate distribution data; this means that we assume that partitions are "ideally" sized (a notion that we will examine more closely in the next subsection). As a result, all partition-based plans end up with just over N tuples in their first partition. Looking at the performance results for Query 1, we see that the Materialize plan ends up being the worst performer among the three partitioning plans because it always materializes 10 MB of temporary partition data, much of which is subsequently not needed. Despite this cost, though, it outperforms the Conventional Sort plan for all values of N except $N = 100,000$ (where their performance becomes essentially the same). The other two partitioning approaches end up providing the best overall performance for this query; both sort only the required amount of data here, neither materializes any excess data in this case, and no excess scans occur, either. These two partitioning plans even outperform the Classic Sort-Stop plan for small N; they use quicksort to sort their first (and only) partition, which makes them slightly less costly here than Classic Sort-Stop, which uses its heap to order the results. Finally, as N increases, the differences between the different approaches diminishes because all of them end up sorting the same amount of data when N reaches 100,000.

3.4 Partitioning and Safety Padding

The preceding experiment provides quite a bit of insight into the relative performance of our old and new Sort-Stop processing schemes for a basic top N query; however, it assumed perfect partitioning. Before we accept its results, we need to explore the sensitivity of the partitioned plans to the number and sizes of partitions used. We need to do so for two reasons. First, we need to understand how to choose the partitioning parameters for each type of plan. Second, we need to find out how costly partitioning errors are so we know how to pad the partition sizes for safety; in practice these values will be selected based on a combination of database statistics and optimizer estimates, and they will therefore be imperfectly chosen.

Figure 5 shows the results of a series of experiments that differ in two ways from the ones we just looked at. Here, we have fixed N at 20,000 and decreased the buffer pool size by a factor of about seven to 600KB (i.e., 150 pages of 4 KB). We use an even smaller buffer pool here so as to stress the need for partitioning; this is similar to scaling up the size of the employee table, but keeps the cost of the experiments within reason. The x-axis in Figure 5 is the number of partitions utilized, and the y-axis is the overall query response time (as before). In this graph, the number of partitions is varied with a "perfect cut", meaning for P partitions and a query stopping cardinality setting of N, we have $P - 1$ "winner" partitions with $N/(P - 1)$ tuples in each one plus one "loser" partition with the leftover tuples in it. For comparison and baseline-setting purposes, in addition to showing the performance of the three partition-based plans, the graph also shows the timing results for the Conventional Sort plan (which do not vary since N is fixed); the Classic Sort-Stop is not applicable here since its heap will not fit in the available buffer memory.

The results shown in Figure 5 provide clear insights into how each of the partition-based plans should be dealt with with respect to choosing the number of partitions. The Reread approach, as one would obviously expect, is very sensitive to the number of partitions used; to avoid costly re-reads, two partitions (one winner, one loser) is the optimal choice. The Materialize approach performs best when the winner tuples are partitioned into memory-sized pieces so that each partition can be sorted in memory in a single pass; this is the case in the figure with five (or more) partitions. The Hybrid approach has a similar optimal point, for the same reason; it outperforms Materialize by about 20 seconds' worth of response time since it does not materialize the 80,000 loser tuples.

Figure 6 shows the results of a series of experiments that explores the question of what happens to the different partitioning plans when the winner/loser cut-point (which is the x-axis in the figure) has been incorrectly estimated; let us call this cut-point C from now on. The goal of this experiment is to obtain insights that we can use to guide the sizing of partitions (e.g., so we know which direction it is better to err in). As before, the query used for the experiments here is Query 1, the buffer pool size is 150 pages, and $N = 20,000$. Learning from the results of Figure 5, Reread has one winner and one loser partition in this experiment, whereas Materialize and Hybrid further partition the winner tuples into memory-sized pieces.

When C is set too low, Figure 6 shows that all of the algorithms get into fairly big trouble. This is because some of the winner tuples, which belong in the query result, end up being placed into the large loser partition. In this case, Reread and Hybrid both have to re-scan the entire 50 MB employee table to get at these tuples, while Materialize must sort even the large loser partition. Under these circumstances, the Conventional Sort plan is able to beat all three of the partitioning plans. Materialize performs the worst here because it does a great deal of expensive reading and writing, and this ends up actually being more costly than a second sequential scan of the employee table.

When C is set too high, Figure 6 shows that the partitioning algorithms still manage to do quite well for Query 1. Materialize has roughly constant cost in this region, as it always materializes all of the "old" employees (independent of C). Hybrid grows slowly more expensive as C increases because it materializes C but not all "old" tuples. Reread grows costly much faster with an increasing cut-point over-estimate. To see why, we need to look at the amount of sorting carried out in the three plans: Reread, which has only one winner partition (with C tuples), must sort this whole winner partition in order to find the top N tuples. On the other hand, Materialize and Hybrid partition their C winner tuples into several small winner partitions so that they need not sort all of these winner partitions in cases in which C is set too high. In any case, the bottom line of this experiment is that all of the partition-based algorithms suffer quite strongly if the number of tuples placed into winner partitions ends up being too small, and suffer much less if too many tuples are classified as likely winners. Thus, it is better to err on the high side, placing too few tuples into the loser partition (i.e., too many into winner partitions), to avoid potential performance instabilities.

3.5 Choosing a Partitioning Vector

Before moving on to other queries, it is worth discussing the issue of how the partitioning vector—i.e., the splitting values that control which attribute value ranges are associated with which partitions—can be chosen for the partition-based plans. The preceding subsection showed us how to choose the partition cardinalities, so the remaining open problem is one of successfully mapping these desired cardinalities back into attribute ranges for the `ORDER BY` attribute(s) of a `STOP AFTER` query. This is essentially the dual of the selectivity estimation problem, which takes a query's attribute value ranges and attempts to estimate cardinalities from those ranges; moreover, it is amenable to the same techniques.

There are essentially two potential answers here. The first is *histograms*, which have already been thoroughly studied (e.g., [PIHS96]) and are available in most database systems today. In particular, equi-depth histograms that provide good accuracy even in the presence of skewed data are well understood [Koo80, PSC84, MD88], and in the case of correlated attributes, multi-dimensional histograms will help [PI97]. Thus, if histograms are available, they can be used to determine partition vectors at query compilation time. If no histograms are available, or it is known that the available histograms provide insufficient accuracy (e.g., for complex queries with many "unpredictable" joins or grouping operations), then *sampling* at run time can be used. Sampling has also been thoroughly studied in the database context (e.g., [LNS90, HS92]), and it has also been shown to be quite cheap [DNS91b]. To conclude, at this point, we rely on existing technology, and the new partition-based approaches we propose in this paper can directly take advantage of any improvements made in this field in the future.

4 Other Examples

Thus far we have seen experimental results (involving our favorite query) that demonstrated the basic tradeoffs related to the alternative range partitioning techniques and that showed some of the advantages of using range partitioning for `STOP AFTER` queries. In this section, we present three additional examples with experimental results that highlight several other advantages of range partitioning. These example queries include a percent query, a nested query, and a join query. Since the tradeoffs between the three alternative partitioning approaches are very similar for all `STOP AFTER` queries, we will focus on Hybrid plans, which use our preferred partitioning method, in this section.

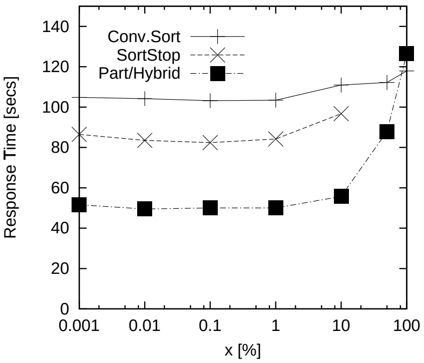


Figure 7: Resp. Time (secs), Query 2
4 MB Buffer, Perfect Partitioning

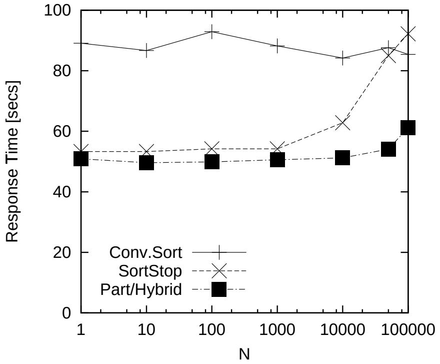


Figure 8: Resp. Time (secs), Query3
4 MB Buffer, Perfect Partitioning

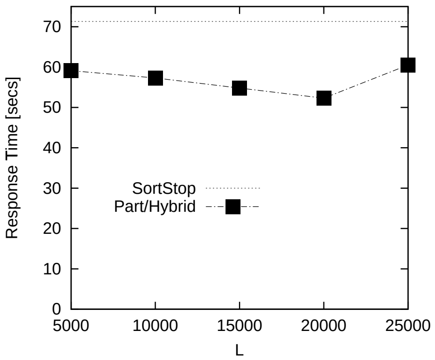


Figure 9: Query 3, Vary L
4 MB Buffer, $N = 20,000$, $C = 25,000$

4.1 Percent Queries

Our first additional example (Query 2) is a so-called *percent query*. The query asks for the $x\%$ highest paid Emps that are more than 50 years old.

Query 2:

```
SELECT      *
FROM        Emp
WHERE       age > 50
ORDER BY    salary DESC
STOP AFTER  x%;
```

Percent queries are interesting because an additional counting step is required in order to find out how many tuples are to be returned. That is, we need to count the number of Emps that are over 50 before we can actually start STOPping (so to speak). Looking back at the plans studied in the previous section, we see that we can directly apply the Conventional Sort plan (Figure 1) to this percent query: the counting step can be carried out as part of the tbscan(Emp,age>50) operator, and the result of this counting step times x% can be propagated to the scan-stop(N) operator before the scan-stop(N) operator starts to produce tuples; this is possible because the sort in between is a pipeline-breaking operator. Likewise, all three partitioning plans of Figure 3 can be applied to our percent query: again, counting can be carried out as part of the tbscan(Emp,age>50) operators and propagated to the scan-stop(N) operators because there exists a pipeline-breaking operator in between (sort and/or part-mat or part-hybrid). The Classic Sort-Stop plan of Figure 1, however, cannot be directly applied in this case. To use a sort-stop operator, we can either read the whole Emp table twice (once to carry out the counting step and once to find the top x%), or we can read the whole Emp table thereby carrying out the counting step and materializing the Emps with age > 50 as a temporary table and then read the temporary table in order to find the top x%. Which one of these two plans is better depends on the selectivity of the age predicate; in our particular example, the second plan is better because only one out of five Emps is older than 50 in our test database. In any case, both of these adjusted Classic-Sort-Stop plans are more expensive than the (inapplicable) Classic Sort-Stop plan of Figure 1.

Figure 7 shows the running times of the Conventional Sort, the adjusted Classic Sort-Stop (with a materialization step), and the partition-based Hybrid plans for this percent query. As described above, the Conventional Sort and Hybrid plans have almost the same running times here as for Query 1 in Section 3, whereas the Classic Sort-Stop plan has a higher cost due to writing and reading temporary results from disk. As a result, the Hybrid plan is the clear winner for all $x \leq 50\%$ for this percent query. Note that the Classic Sort-Stop plan cannot be applied for $x \geq 50\%$ because its memory requirements then exceed 4 MB.

To find the proper partitioning vector for the Hybrid plan for this query, the observations of Section 3.4 essentially still apply. That is, we should partition the data into memory-sized portions and "play it safe" by materializing too many rather than too few Emp tuples in the part-hybrid operator.

4.2 STOP in a Subquery

In the second example of this section (Query 3), we consider a query that has a STOP in a subquery. Our example query asks for the average salary of the N best paid Emps with age > 50.

Query 3:

```
SELECT AVG(e.salary)
FROM   (SELECT      salary
        FROM        Emp
        WHERE       age > 50
        ORDER BY    salary DESC
        STOP AFTER  N) e;
```

Both the Conventional Sort and the Classic Sort-Stop plan of Figure 1 can be applied to this query; they simply need an aggregate operator at the top in order to compute the average. These traditional plans, however, perform a great deal of wasted work since they produce their output in salary order and this ordering is not needed to compute the average. With a partition-based plan, most of this sorting can be avoided by partitioning the Emps into three partitions: one partition containing the top L Emps (L slightly smaller than N), one partition containing the next M Emps such that M is small and $L + M \geq N$, and one partition with the all of the other loser Emps. In this case, only the M Emps in the second partition need to be sorted in order to find the $N - L$ highest paid Emps in that partition.

Figure 8 shows the running times of the three alternative plans for this query. We can see that the Conventional Sort plan again has the highest cost because it sorts all 100,000 `Emps` with `age > 50`, independent of N. Also, as in the previous experiments, the cost of the Classic Sort-Stop plan increases with N and is in between the Hybrid and Conventional Sort plans. What makes Query 3 and this experiment special is that the cost of the Hybrid plan is almost constant here because Hybrid sorts very few `Emps`, independent of N; only for very large N does the cost of the Hybrid plan slightly increase, due to materializing many `Emp` tuples. As a result, the differences in cost between the Classic Sort-Stop and Hybrid plans increase sharply with N, and the Hybrid plan outperforms the Conventional Sort plan even for $N = 100,000$. It should be noted that the cost of the Conventional Sort plan is lower for this query than in all previous experiments because this query can be evaluated using only the `salary` column of `Emps` (i.e., the other columns are projected out after the `tbscan`), permitting the `sort` to be carried out in one pass in memory. Similarly, the Classic Sort-Stop plan can be used for all N for this query without exhausting the buffer space.

It is somewhat trickier to find a perfect partitioning vector for this query than for Queries 1 and 2. If we set $C = L + M$ (C is the "cut-point" between winners and losers as in Section 3.4), then we need to make sure that $L \leq N$ in addition to $C \geq N$ and C as small as possible. In other words, here we need to find a good *left* cutting point, L, in addition to a good *right* cutting point, C, whereas we only needed to find a good right cutting point for Queries 1 and 2. Figure 9 shows the sensitivity of the cost of the Hybrid plan towards cases in which L is set imperfectly for $N = 20,000$ and $C = 25,000$. Obviously, the Hybrid plan performs best if L is close to N. However, the figure shows that the penalty for a poor setting of L is not severe (20% at most) due to the fact that the additional work is proportional to the number of misclassified tuples (i.e., errors here don't cause entire additional partitions to become involved in the query plan); in any case the Hybrid plan outperforms both traditional plans. (Figure 9 shows the cost of the Classic Sort-Stop plan, the better of the two traditional plans, as a baseline.)

4.3 Join Queries

The last example query of this section involves a join; this example shows that partitioning becomes even more attractive for more complex queries. The query asks for the N highest paid `Emps` that have `age > 50` and that work in a `Dept` with `budget` $>$ `1,000`.

Query 4:

```
SELECT      *
FROM        Emp e, Dept d
WHERE       age > 50
            AND d.budget > 1000
            AND e.works_in = d.dno
ORDER BY    salary DESC
STOP AFTER  N;
```

Figure 11 shows two traditional plans for this query. The first plan is based on (conventionally) sorting the `Emp` table into `salary` order and then probing the top `Emps` one by one in order to find out whether they work in a `Dept` with a high `budget` (i.e., it uses an index nested-loop join). The second plan carries out the join first, in order to find all `Emps` that work in a `Dept` with a high `budget` (Grace-hash join is best for this purpose in our test database), and then it finds the N highest paid of these `Emps` using a `sort-stop` operator. As an alternative to these two traditional plans, Figure 12 shows two partitioning-based plans for this query. The idea here is to partition the `Emp` table before the join, and then to join one `Emp` partition at a time with the `Dept` table until at least N `Emps` that survive the join have been found. Thus, just as partitioning was used in the previous examples to avoid unnecessary sorting work, partitioning is utilized in these two join plans to avoid unnecessary sorting *and* join work. The difference between these two plans is that the first one uses index nested-loops for the join, whereas the second one uses hashing. Note that for small N, the hash join of the Part+HJ plan can be carried out in one pass if the `part-hybrid` operator partitions the data into memory-sized portions.

Figure 10 shows the running times of the four plans, varying N and using our test database in which all `Depts` actually have a `budget > 1,000`. We see immediately that the partitioning plans clearly outperform the two traditional plans. The Sort+NLJ plan has the highest cost, independent of N, because it always sorts all 10 MB of `Emps` with `age > 50`. For $N > 1000$, it has extremely high costs because, in addition to the expensive `sort`, the `NLJ` becomes very costly because many `Emp` tuples generate probes, resulting in an excessive amount of random disk I/O. The GHJ+Sort-Stop plan has low sorting costs for small N, but it has poor performance because it performs a full-fledged Grace-hash join. For $N \geq 50,000$, the GHJ+Sort-Stop plan is again inapplicable because the buffer requirements of the `sort-stop(N)` operator exceed the limit of 4 MB. (Replacing the `sort-stop(N)` operator by a conventional `sort` and a `scan-stop(N)` operator would yield an execution time of about 250 secs here.) Both partitioning variants avoid unnecessary sorting and joining of `Emp` tuples. The Part+NLJ plan performs best for small N, but its performance deteriorates for $N > 1,000$ due to the high cost of the NLJ, just as in the Sort+NLJ plan. The Part+HJ plan shows better performance in these cases because hash joins are better than index nested-loop joins when both input tables are large.

In terms of sensitivity, the points mentioned for Queries 1 and 2 still basically apply; we should make sure that the first partition contains all of the `Emp` tuples needed to answer the query. We must keep in mind, however, that for the Part+HJ plan, the penalty for setting the "cut-point" too low is higher than for the partitioning plans for the simple sort queries because a restart involves not only re-scanning the `Emp` table, but also re-scanning the `Dept` table in the Part+HJ plan. Since the Part+NLJ plan never actually scans the `Dept` table, the Part+NLJ plan does not pay this additional penalty for restarts.

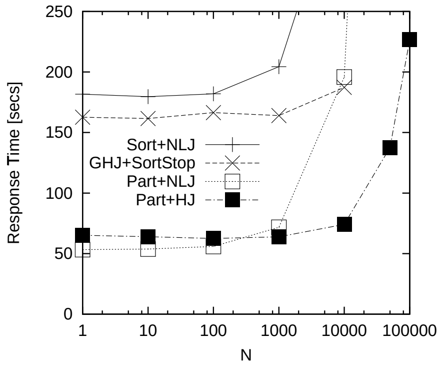
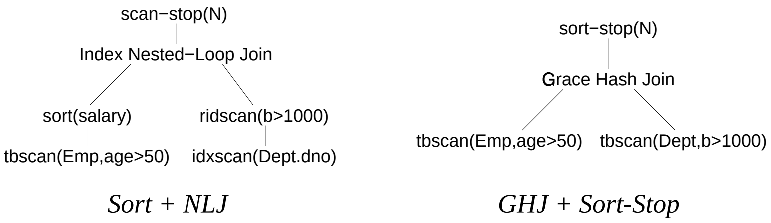


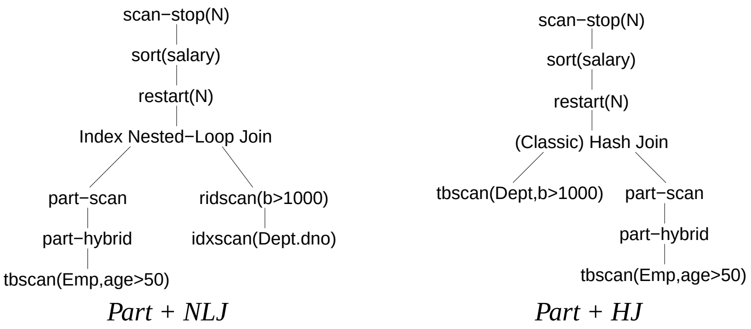
Figure 10: Resp. Time (secs), Query 4
4 MB Buffer, Perfect Partitioning



Sort + NLJ

GHJ + Sort-Stop

Figure 11: Traditional Plans for Query 4



Part + NLJ

Part + HJ

Figure 12: Range Partitioning Plans for Query 4

5 Other Techniques

We have seen that range partitioning can be very helpful to improve the response time of several different types of `STOP AFTER` queries. In this section, we will show how two other techniques can be applied to evaluate `STOP AFTER` queries. The first technique is also based on partitioning, but it is based on using ordered indexes (e.g., B^+ trees) to partition the data. The second technique is based on using semi-joins to reduce the size of temporary results.

5.1 Partitioning with Indexes

Let us return to Query 1, which asks for the N `Emps` with the highest `salary` and `age > 50`, and see what happens when we have a B^+ tree on `Emp.salary`. The Traditional Index-Scan plan that executes this query using the `Emp.salary` index was shown in Figure 1 and discussed in Section 2.1; it reads the RIDs of the `Emps` one at a time in `salary` order from the index, then fetches the `age`, `address`, etc. fields and applies the `age` predicate, until the top N *old* `Emps` have been found. In Section 2.1, we noted that this plan would be very good if the `Emp.salary` index is clustered or N is very small, but that it would have a high cost if N is large and/or the `age` predicate filters out many high paid `Emps` because, in this case, the `ridscan` operator would lead to a great deal of random I/O and many page faults for rereading pages of the `Emp` table if the buffer is too small to hold all of the relevant pages of the `Emp` table.

For large N and unclustered indexes, we can do better by using, of course, partitioning. The idea is to read the RIDs of the top N' `Emps` from the index, sort these N' RIDs in *page id* order, do the `ridscan` with the predicate, re-sort into `salary` order, and cut off the top N tuples or repeat if less than N of the top N' `Emps` have `age > 50`. Similar RID sorting ideas are known as RID-list processing and have been commonly exploited in text databases (e.g., [BCC94]) and set query processing (e.g., [Yao79]), but they can only be applied in the `STOP AFTER` context if they are combined with partitioning. The beauty of this *Part-Index* approach is that the `ridscan` operator becomes quite cheap since it reads the `Emp` pages sequentially and reads no `Emp` page more than once from disk. On the negative side, this approach involves two sorting steps. If N and N' are small, however, these sorts are fairly cheap because they can be carried out in one pass in memory.

Figure 13 shows the running times of the Traditional Index-Scan plan and a Part-Index plan for Query 1, varying N. As baselines, the figure also shows the running times of the Hybrid plan that does not use the `Emp.salary` index (as in Figure 4) and the “ideal” running time for Query 1 generated by running the Traditional Index-Scan plan on a special version of our test database in which the `Emp.salary` index is clustered. We see that the Part-Index plan clearly outperforms the Traditional Index-Scan plan for a large range of N. While the Traditional Index-Scan plan is only attractive for $N \leq 100$, the Part-Index plan shows almost “ideal” performance up to $N = 10,000$. (After that its sorts become too expensive.) Only for $N = 10$ does the Traditional Index-Scan plan slightly outperform the Part-Index plan (0.9 secs vs. 1 sec).

The right setting of N', of course, depends upon both N and the selectivity of the `age` predicate. In this example, N' should be set to $5 * N$ because every fifth `Emp` is older than 50 in our test database and the values of the `salary` and `age` columns are not correlated. It should be noted, however, that the penalty for restarts in the Part-Index plan is very low: rather than re-scanning the entire `Emp` table, a restart simply involves continuing the `Emp.salary` index scan and fetching the next N' `Emp` tuples.

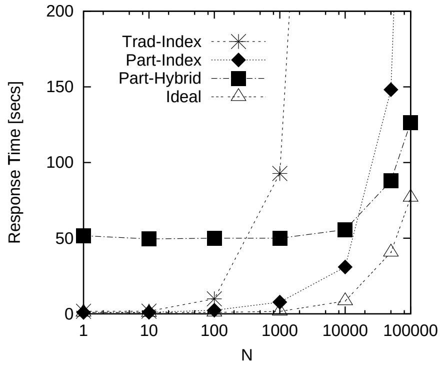


Figure 13: Resp. Time (secs), Query 1
4 MB Buffer, Unclustered `Emp.salary` Index

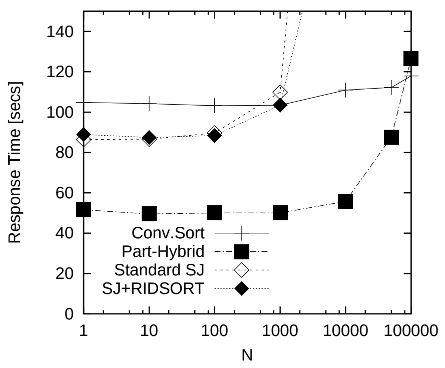


Figure 14: Resp. Time (secs), Query1
4 MB Buffer, Semi-join Plans

5.2 A Semi-Join-Like Technique

The idea of semi-joins is to reduce the cost of I/O intensive operations, such as sorts, joins, and group-by operations, by projecting out all but those columns actually needed for an operation; doing so reduces the size of the temporary results that need to be read and/or written to disk or shipped through a network. The disadvantage of semi-joins is that columns that were projected out must be re-fetched (using a `ridscan` operator) after the operation in order to carry out subsequent operations or because they are part of the query result. Semi-join techniques have been extensively used in the early distributed database systems (e.g., [BGW$^+$81]), but they have not been widely used in centralized database systems. One reason for this is the potentially prohibitively high and unpredictable costs of re-fetches, though the cost of the re-fetches can be reduced with the same RID sorting trick described in the previous subsection. What makes semi-join-like techniques attractive for `STOP AFTER` queries is that the costs of the re-fetches are limited and can be predicted accurately during query optimization if the re-fetches are always carried out at the end of query execution: if a query asks for the top N tuples, then at most N re-fetches are required at the end.

We studied two different semi-join-like plans for Query 1. In both plans, the sorting of the `Emp` tuples can be carried out in one pass in main memory because only the RIDs and the *salary* fields of the `Emp` tuples are kept. The difference between the two plans is that the first plan which we call the *Standard SJ Plan* does not apply the RID sorting trick described in the previous subsection in order to improve the re-fetches, whereas the second plan, the *SJ+Ridsort Plan*, does apply this trick. We ran both plans with varying N, and Figure 14 shows the results of these experiments. The figure also shows the results of the Conventional Sort and Hybrid plans of Section 3 as baselines. First, we see that the impact of the RID sorting trick is less pronounced than in the previous experiments with the Part-Index plan. The reason is that in both semi-join plans, the `Emp` table has already been read once, and the *old* `Emps` have already been filtered out, so the `ridscan` gets more hits in the buffer pool and is applied to fewer tuples. Second, we observe that for small N, the two semi-join plans indeed outperform the Conventional Sort plan, while for $N \geq 1000$, the performance of the semi-join plans deteriorates due to their high re-fetching costs. Finally, we see that the semi-join plans are clearly outperformed for all N by the partition-based Hybrid plan; like the semi-join plans, the Hybrid plan of Section 3 carries out its `sort(salary)` in one pass (for $N \leq 10,000$), and the Hybrid plan has the additional advantage of sorting slightly more than N rather than all 100,000 `Emps` with `age > 50`. We note that it is not too difficult to find other example queries where a semi-join plan would actually outperform a partitioning plan (e.g., for very large and highly selective joins with small N); sometimes the best plan to execute a query may be a combination of partitioning and semi-joins.

6 Conclusions

In this paper, we presented several new strategies for efficiently processing `STOP AFTER` queries. These strategies, based largely on the use of range partitioning techniques, were shown to provide significant savings for handling important classes of `STOP AFTER` queries. We presented examples including basic “top N” queries, percent queries, subqueries, and joins; we saw benefits from the use of the new partitioning-based techniques in each case due to the reduction in wasted sorting and/or joining effort that they offer. We showed that range partitioning can be useful for indexed as well as non-indexed plans, and we also showed that semi-join-like techniques can provide an additional savings in some cases.

There are several areas that appear fruitful for future work. One area of interest is `STOP AFTER` query processing for parallel database systems. Our techniques should be immediately applicable there, and they may offer even greater benefits by reducing the amount of data communication required to process such queries. Another avenue for future investigation would be experimentation with the effectiveness of histogram and/or sampling techniques for determining the partitioning vector entries for `STOP AFTER` queries on real data sets.

Acknowledgments

We would like to thank Paul Brown of Informix, Jim Gray of Microsoft, Anil Nori of Oracle (and DEC, in a previous job), and Donovan Schneider of RedBrick Systems for informing us about the support for *top N* query processing in their respective companies' current database products.

References

[AHU83] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA, USA, 1983.

[BCC94] E. Brown, J. Callan, and B. Croft. Fast incremental indexing for full-text information retrieval. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 192–202, Santiago, Chile, September 1994.

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

[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.

[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, AZ, USA, May 1997.

[DG92] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[DNS91a] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of non-equijoin algorithms. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 443–452, Barcelona, Spain, September 1991.

[DNS91b] D. DeWitt, J. Naughton, and D. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proc. of the Intl. IEEE Conf. on Parallel and Distributed Information Systems*, pages 280–291, Miami, Fl, USA, December 1991.

[DNSS92] D. DeWitt, J. Naughton, D. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 27–40, Vancouver, Canada, August 1992.

[HS92] P. Haas and A. Swami. Sequential sampling procedures for query size estimation. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 341–350, June 1992.

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

[Koo80] R. Kooi. *The optimization of queries in relational databases.* PhD thesis, Case Western Reserve University, September 1980.

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

[LNS90] R. Lipton, J. Naughton, and D. Schneider. Practical selectivity estimation through adaptive sampling. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 1–11, Atlantic City, USA, April 1990.

[MD88] M. Muralikrishna and D. DeWitt. Equi-depth histograms for estimating selectivity factors for multidimensional queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 28–36, Chicago, IL, USA, May 1988.

[PI97] V. Poosala and Y. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 486–495, Athens, Greece, August 1997.

[PIHS96] V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 294–305, Montreal, Canada, June 1996.

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

[WKHM98] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. 1998. Submitted for publication.

[Yao79] S. Yao. Optimization of query evaluation algorithms. *ACM Trans. on Database Systems*, 4(2):133–155, 1979.