Stop-and-Restart Style Execution for Long Running Decision Support Queries

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

Raghav Kaushik
Microsoft Research
skaushi@microsoft.com

Abhijit Pol
University of Florida
apol@cise.ufl.edu

Ravi Ramamurthy
Microsoft Research
ravirama@microsoft.com

ABSTRACT

Long running decision support queries can be resource intensive and often lead to resource contention in data warehousing systems. Today, the only real option available to the DBAs when faced with such contention is to carefully select one or more queries and terminate them. However, the work done by such terminated queries is entirely lost even if they were very close to completion and these queries will need to be run in their entirety at a later time. In this paper, we show how instead we can support a Stop-and-Restart style query execution that can leverage partially the work done in the initial query execution. In order to re-execute only the remaining work of the query, a Stop-and-Restart execution would need to save all the previous work. But this approach would clearly incur high overheads which is undesirable. In contrast, we present a technique that can be used to save information selectively from the past execution so that the overhead can be bounded. Despite saving only limited information, our technique is able to reduce the running time of the restarted queries substantially. We show the effectiveness of our approach using real and benchmark data.

1. INTRODUCTION

Decision support queries can be long running. For example, recent TPC-H [17] benchmark results show that these queries might take even hours to execute on large datasets. When multiple long running queries are executed concurrently, they compete for limited resources including CPU, main memory, and workspace area on disk used to store temporary results, sort runs and spilled hash partitions. Contention for valuable resources can substantially increase the execution times of the queries. It is possible to suspend the execution threads of one or more low-priority queries and resume them at a later time. The main problem with this approach is that suspending the execution of a query only releases the CPU resources; the memory and disk resources are still retained until the query execution thread is resumed. Thus, the only real option available today in order to release *all* resources is to carefully select one or more queries (based on criteria such as the importance of the query or the amount of resources used by it or progress information [3][13]) and terminate them. Terminating one or more queries releases all resources allocated to them, which can be used to complete other queries.

In today's database systems, the work done by such terminated queries is entirely lost even if they were very close to completion and these queries will need to be run in their entirety at a later time. In this paper we show how instead we can support a *Stop-and-Restart* query execution that can leverage partially the work done in the initial query execution to reduce the execution time if we were to restart it.

Any attempt to save and reuse all intermediate results potentially requires very large memory and/or disk resources (for instance, hash tables in memory, sort runs in disk, etc.) in the worst case, amounting to significant overheads. Therefore, in this paper we propose Stop-and-Restart query execution that is constrained to save and reuse only a *bounded* number of records (intermediate records or output records) thus releasing all other resources. Such an approach is attractive since it limits the resources retained by a query that has been terminated. We refer to this as the *bounded query checkpointing* problem. Our solution is based on choosing a subset of records to save during normal execution and then skipping the corresponding records when performing a scan during restart. We introduce a generalization of the scan operator called *skip-scan* to facilitate such a restart technique. We demonstrate that in order to obtain significant speed up, it is necessary to carefully select the subset of records to save. The key challenge we address is to select this set of records online as query execution proceeds, since we have no knowledge of when or if at all the query is going to be terminated. Our experiments based on a prototype built by modifying Microsoft SQL Server 2005 indicate that there are many cases where bounded query checkpointing yields significant benefits even when a relatively small number of records are saved. Of course, we get higher benefits when more records can be saved.

The primary target for Stop-and-Restart style execution is decision-support queries issued in a data-warehousing environment. We assume that the database is read only except for a batched update window when no queries are executed. Finally, although saving and reusing intermediate results is reminiscent of the techniques used for dynamic query optimization [1][6][15], the constraint of saving and reusing only a *bounded* number of records is unique to our problem.

The outline of the paper is as follows. In Section 2 we review some preliminary concepts and also present desirable properties of Stop-and-Restart style query execution. Section 3 discusses Stop-and-Restart execution for query plans consisting of a single pipeline. In this section, we introduce how Stop-and-Restart query execution can be supported in the framework of query processing engine of relational database systems. Based on a GetNext model

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Database Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permissions from the publisher, ACM.
VLDB '07, September 23-28, 2007, Vienna, Austria.
Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

of work for query execution, we also present an algorithm to compute the set of intermediate results to save and reuse for the case of single pipeline execution plans. In Section 4, we extend our algorithms to work for more complex execution plans. We present an experimental evaluation of our prototype system in Section 5 and present discussions on important extensions in Section 6. Section 7 presents related work and Section 8 concludes the paper.

2. PRELIMINARIES

2.1 Pipelines and Plans

The Stop-and-Restart style query execution is described around *query execution plans*. A query execution plan is a tree where nodes of the tree are physical operators. Each operator exposes a GetNext interface and query execution proceeds in a demand driven fashion [7]. In this paper, for ease of exposition we do not consider parallel query execution plans.

An operator is a *blocking operator* if it produces no output until it consumes at least one of its inputs completely. The Hash join is an example of blocking operator. The probe phase cannot begin until the entire build relation is hashed.

A *pipeline* is a maximal subtree of operators in an execution plan that execute concurrently. The examples in Figure 1 illustrate plans that execute in a single pipeline. Every pipeline has one or more *source nodes*, the operator that is the source of the records operated upon by remaining nodes in the pipeline. The Table Scan A in Figure 1(a) and Index Scan A in Figure 1(b) are examples of source nodes. We discuss execution plans consisting of multiple pipelines in Section 4.

2.2 Modeling Work during Query Execution

We require a way to measure the amount of work done during query execution. One natural candidate is the optimizer cost model; however we require a more light-weight alternative. In this paper we use the *total number of GetNext calls* measured over all the operators to model the work done during query execution. While a weighted aggregation of GetNext calls is more appropriate for complex queries involving operations such as subqueries and UDFs, we use the counts as a first step. This is consistent with recent work on progress estimation for queries which also employs similar schemes to model the work done during query execution [3][13][14][4]. Our experiments in Section 5.1 indicate that the improvement as measured by the GetNext model is almost the same as that in execution times.

2.3 Stop-and-Restart Style Query Execution

Stop-and-Restart style query execution involves two distinct phases – the *initial* run which is the first query execution until it is terminated, and the *restart* run which is the re-execution of the same query at a later time. We are allowed to save some state during the initial run which can be utilized during the restart run. When the query is killed, this state is saved coupled with a modified execution plan utilizing it. During the restart run, the modified plan (referred to as a *restart plan*) is executed. In this paper, we only consider saving intermediate results generated during the initial run. We discuss other candidates for saving such as the internal state of operators in Section 6. Also, while our approach can be extended to cover the case where the storage constraint is given in terms of number of bytes, in this paper we consider the constraint specified in terms of number of records for simplicity.

We now discuss desirable properties of a Stop-and-Restart style execution.

Correctness: The restart plan must be equivalent to the original query plan.

Low Overhead: There are two forms of overhead in a Stop-and-Restart framework. One is the monitoring overhead incurred when the query is not terminated. Clearly, the performance in this case should be comparable to normal query execution. Secondly, we have the stop-response-time, which is the time taken to terminate the query. The process of query termination must be fast, which constrains the number of records we can save. This is a critical design point for the viability of our approach.

Generality: A Stop-and-Restart framework should be applicable to a wide range of query execution plans.

Efficiency of Restart: The sum of the execution time before the query is stopped and the execution time after it is restarted should be as close as possible to the execution time of uninterrupted query execution. Thus, our main performance metric is how much of the work done during the initial run can be saved during the restart run.

3. SINGLE PIPELINE QUERY PLANS

We consider pipelines that consist of a single source node and where the results of the pipeline are obtained by invoking the operator tree on each source record in order and taking their union. The pipelines in Figure 1 fall in this class – the source nodes are shaded. Result records are generated at the *root* node of the pipeline. At any point in execution, it is meaningful to talk about the *current source record* being processed in the pipeline. There are pipelines having operators such as Top, Merge-Join that do not fall in this class; our techniques however are applicable to such pipelines also. We extend our solution to query execution plans consisting of multiple pipelines in Section 4.

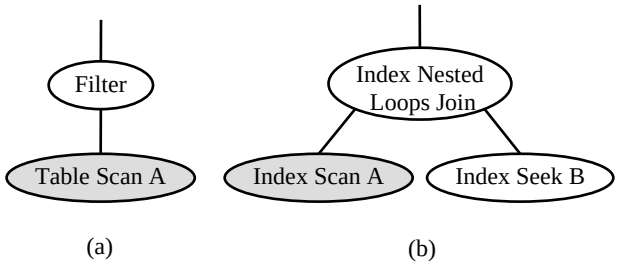

Figure 1. Examples of a single pipeline query execution plans.

3.1 Primitives for Stop-and-Restart Execution

In this section, we outline the primitives for supporting Stop-and-Restart execution.

Rids at the Source Node

We assume that each source record has a unique identifier *rid*. This can be implemented say by adding the primary key value to the key of a clustering index. For instance, Microsoft SQLServer adds the primary key to any index key. Without loss of generality, we assume that *rid*s are numbered 1,2,3… in the order in which

they are scanned. For ease of exposition, we assume a special rid value 0 indicating the beginning of the table. We use the notation (LB,UB) to denote all source records with rids between, but not including LB and UB, whereas [LB,UB] also includes LB and UB. We also assume that for any intermediate record r, we can obtain the rid for the corresponding source record, denoted as *Source(r).*

Skip-Scan Operator

The simplest stop-restart technique is to *save* all result records generated during the initial run at the root of the pipeline. For example, for the plan shown in Figure 1(a), we save all records returned by the Filter operator. During the restart run, the goal is to avoid re-computing these saved results. We accomplish this by introducing the notion of *skipping* – we skip the corresponding source records when scanning the source in the restart run.

We introduce a generalized version of a scan operator that can be used to support this. It takes two rids LB < UB as an input. It scans all records in the source node up to and including LB and resumes the scan from the record with rid UB (included in the scan), skipping all records in between. We refer to this primitive as the *skip-scan* operator.

The skip-scan operator can be built on top of existing operators such as Table Scan and Clustered Index Scan utilizing the random access primitives from the storage manager. For instance, in a Clustered Index Scan, we seek to the UB value using the key (we assume they are unique). In the case of Heap File Scan, we remember the page (pageid, slotid) to resume the scan from. In general, the skip-scan operator can be extended to skip multiple portions of the source node. In this paper, we focus on skipping a single contiguous range.

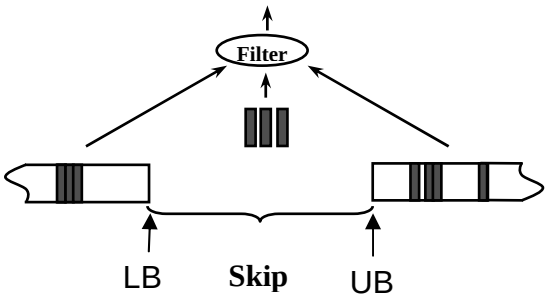

Figure 2. Skip-Scan Plan

Saving and Returning Records

We extend all operators with the ability to save a sequence of records. This logic is only invoked at the root of the pipeline, which is marked at compilation time. If and when the query is terminated, a restart plan that uses this sequence of records is saved where the source node is replaced with a corresponding skip-scan operator.

Any pair of rids LB < UB (at the source node) identifies a restart plan RPlan(LB,UB) as follows. We replace the scan of the source node with a skip-scan seeded with LB and UB and cache the results generated by records in the region (LB,UB) at the root of the pipeline.

Now we explain the execution of the restart plan. Consider the point where the skip-scan operator has returned the source record corresponding to LB. At this point, similar to the end-of-stream (EOS) message that a scan operator sends at termination, the skip-scan operator sends an end-of-LB (EOLB) message before it skips to the UB. On receiving the EOLB message the pipeline root returns the saved records, after which it makes GetNext calls to its child operator as usual. Figure 2 illustrates an example of a restart plan. The Filter operator is the root of the pipeline which returns the three saved records on receiving the EOLB message from the skip-scan operator.

3.2 Restart Plans and Their Benefit

Instead of reasoning in terms of cost, we reason in terms of *benefit*. The *benefit* of a restart plan is the number of GetNext calls skipped, i.e., the difference between the number of GetNext calls completed while executing the original plan and the restart plan. For ease of exposition, we ignore the GetNext calls involved in returning the results cached at the root of the pipeline of a restart plan. However, all our results extend even if we count these calls.

Since we cache result records at the root of the pipeline, we search the space of restart plans by examining result records at the root. For a window W consisting of contiguous records $r_{i-1},\ldots,r_{i+j}$ ($j \geq 0$) at the root of the pipeline, we use the corner records r_{i-1} and r_{i+j} to derive a restart plan as follows. The set of result records excluding the two corners, that is $r_i,\ldots,r_{i+j-1}$ is called the *candidate set* underlying W with size *j*. By setting LB = *Source*(r_{i-1}) and UB = *Source*(r_{i+j}) and saving the candidate set, we obtain a *candidate restart plan*.

However, the candidate restart plan is not necessarily equivalent to the original query plan, as illustrated by the following example. Suppose we are executing an Index Nested Loop Join between Tables A and B as shown in Figure 3. Consider the sliding window of three result records r_0 = (1,1), r_1 = (2,2) and r_2 = (2,2) shown shaded in the figure. The restart plan corresponding to this is defined by LB = 1 and UB = 2 thus leading to no record being skipped. The candidate set however has the single record r_1 = (2,2) which implies that this restart plan is incorrect. Such duplication happens if and only if *Source*(r_{i-1}) = *Source*(r_i) or *Source*(r_{i+k}) = *Source*(r_{i+k-1}).

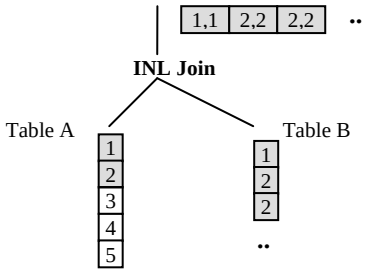

Figure 3. Example of Result Window that is not Skippable

Result windows where *Source*(r_{i-1}) ≠ *Source*(r_i) and *Source*(r_{i+k}) ≠ *Source*(r_{i+k-1}) are called *skippable*. Thus, Figure 3 illustrates an example window that is not skippable.

The candidate restart plan corresponding to a skippable window W is denoted as RPlan(W) and the benefit of RPlan(W) as *benefit*(W).

We need an additional mechanism to handle certain corner cases. We assume two "dummy" result records appearing at the root of the pipeline: *begin* associated with the iterator's Open call, and *end* associated with the call to Close. *Source(begin)* is defined to

be 0. *Source(end)* is set to be the current source record being processed at the point of termination. Consider the query plan in Figure 1(a). Suppose that at the point of termination, no records have been output by the filter operator. In this case, we can skip the entire scan until this point. However, a candidate restart plan is only defined for windows that have at least two corner records. We need *begin* and *end* to capture such cases.

3.3 Bounded Query Checkpointing

We can use the primitives introduced above to save all result records but this incurs unbounded overhead since the number of results generated can be large. We control the overhead by constraining the number of records that can be saved.

A skippable window W of result records is said to be *bounded* if its candidate has size at most *k*. The corresponding restart plan is also said to be *bounded*. In general, RPlan(LB,UB) is said to be bounded if the number of results cached is at most *k*.

The *bounded query checkpointing* problem is the following online problem. We are given a pipeline P and a budget of *k* records. At any point in execution where the current source record being processed has identifier *id*, the goal is to maintain a bounded restart plan equivalent to P that yields the maximum benefit among all bounded restart plans RPlan(LB,UB) with LB < UB ≤ *id*. This is an online problem since we do not know when the query is going to be terminated.

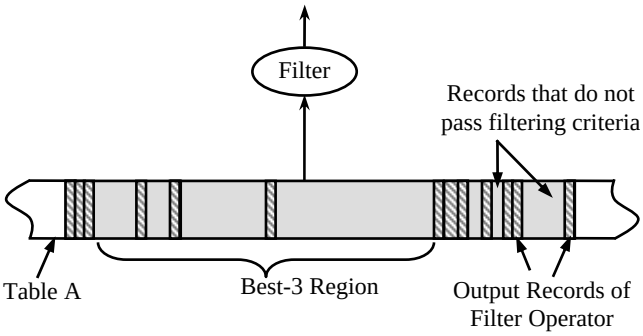

Figure 4. Optimal Bounded Restart Plan.

Figure 4 illustrates an example for the query plan in Figure 1(a) where the budget *k* is three. The records that satisfy the filter predicate are marked out (other records are in the gray region). Suppose the query is terminated after all the records shown in the Figure are processed. The label "Best-3 Region" shows the region that is skipped in the optimal restart plan.

There is an inherent tradeoff between the number of records we can cache and the benefit we can obtain during restart. For a given budget *k*, there are cases where the maximum benefit we can obtain is limited, independent of the specific algorithm we use. Consider the query `select * from T` that scans and returns all records in T. Any algorithm can skip at most *k* records in the scan. If *k* is small compared to the cardinality of T, then most of T has to be scanned during restart.

However, in practice, there are cases where even a small value of *k* can yield significant benefit provided we carefully choose which *k* records to save. This holds even when the budget *k* is 0. For example, in Figure 4, we can skip the region between any two successive source records that satisfy the filter predicate. We now discuss an experiment that illustrates this point.

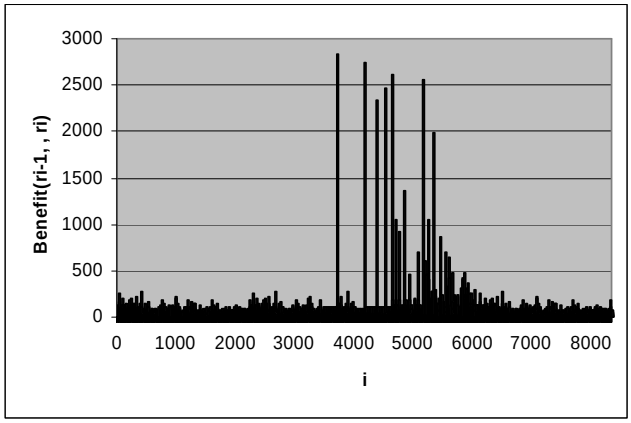

Figure 5. Benefit(r_{i-1}, r_i) vs. *i*.

We choose a query over the TPC-H data that selects all rows from the lineitem table that satisfy the predicate `l_shipdate <= '1996-12-01'`. The execution plan for this query is identical to the plan shown in Figure 1(a). We execute this query over the TPC-H dataset where the data distribution is skewed (refer to Section 5 for details). Figure 5 plots the benefit of the result window (r_{i-1},r_i) (Y-axis) against *i* (this corresponds to setting the budget to 0). We note that there is significant non-uniformity in the graph. Most of the benefit is clustered in a range of about 2000 records between r_{4000} and r_{6000}, whereas the lineitem table has 6 million records. Now suppose that the budget *k* is 2000. The benefit of saving the 2000 records between r_{4000} and r_{6000} can be obtained by aggregating the corresponding benefits from Figure 5. This benefit is significant and is much higher than the aggregate corresponding to an arbitrary subset of 2000 records.

3.4 *Opt-Skip* Algorithm

We now present the *Opt-Skip* algorithm for the bounded query checkpointing problem. This algorithm runs at the root node of the pipeline and considers various restart plans identified by maintaining a sliding window of result records.

A naïve strategy suggested by the problem statement in Section 3.3 enumerates all bounded restart plans as result records arrive at the pipeline root. However, it is not necessary to enumerate all bounded restart plans. Observe that if we have two restart plans RP_1 = RPlan(LB_1,UB_1) and RP_2 = RPlan(LB_2,UB_2) where $LB_1 \leq LB_2$ and $UB_1 \geq UB_2$, then *benefit(RP_1)* ≥ *benefit(RP_2)*. Thus, it suffices to consider only *maximal* restart plans defined to be plans (1) which are bounded and (2) where decreasing LB or increasing UB violates the bound.

We capture this idea in our algorithm by considering *maximal* skippable windows of result records. Given a window W, an *extension* is any window W' that has W as a proper sub-window (so W' must have at least one more record than W). A skippable window W is said to be *maximal* if it is bounded and it has no skippable extension that is also bounded. It is not hard to see that maximal restart plans correspond to maximal skippable result windows and vice versa.

Our algorithm outlined in Figure 6 enumerates restart plans corresponding to maximal skippable windows of result records.

The constraint on the bound is met by maintaining a sliding window W of k+2 result records. The current window W is not necessarily skippable, which is why the method *FindSkippable* is invoked to find its largest sub-window that is skippable. Consider the current window of size k+2. Let it be W = $r_{i-1},\ldots,r_{i+k}$. If W is not skippable, then the largest skippable sub-window can be found by finding the last record in W with source *Source*(r_{i-1}) and the first record with source *Source*(r_{i+k}). A skippable sub-window exists if and only if *Source*(r_{i-1}) ≠ *Source*(r_{i+k}).

```
/* W = current window, k= total budget */
/* BestW = best window */
Algorithm Opt-Skip
   BestW = Null
   W = Null
   For Each intermediate record r_i do:
      Append r_i to W
      If W.Size() > k+2 then
         W = last k+2 records in W
      SkippableW = FindSkippable(W)
      If Benefit(SkippableW) > Benefit(BestW) then
         BestW = SkippableW

Algorithm FindSkippable
Input: W = r_{i-1},...,r_{i+j}.
   If (Source(r_{i-1}) = Source(r_{i+j}))
      Return Null
   Find the least j1 such that Source(r_{i-1}) ≠ Source(r_{i-1+j1})
   Find the least j2 such that Source(r_{i+k-j2}) ≠ Source(r_{i+k})
   Return the window (r_{(i-1+j1)-1},...,r_{(i+k-j2)+1})
```

Figure 6. Algorithm *Opt-Skip*

The other aspect of our algorithm is the computation of the benefit of a restart plan. This is computed online as follows. For result record r_i, let $GN_{\leq}(r_i)$ be the total number of GetNext calls issued in the pipeline until the point record r_i was generated at the root. Let $GN(r_i)$ denote the number of GetNext calls needed to generate r_i at the root beginning by invoking the operator tree on record *Source*(r_i) from the source. For a skippable window of result records W= $r_{i-1},\ldots,r_{i+j}$, we can show that the benefit is

$benefit(W) = GN_{\leq}(r_{i+j}) - GN_{\leq}(r_{i-1}) - GN(r_{i+j})$

This formula enables us to compute the benefit in an online fashion. In our implementation, we focus on pipelines consisting of operators such as filters, index nested loops and hash joins where $GN(r_i)$ is the number of operators in the pipeline. For such pipelines, maximizing the benefit as stated above is equivalent to maximizing $GN_{\leq}(r_{i+j}) - GN_{\leq}(r_{i-1})$. The null window referenced in Figure 6 is defined to have a benefit of 0.

If the number of candidate records returned at the pipeline root is less than or equal to the budget k, then we save all of them. Whenever we find a set of result records in the current window that is skippable and has a higher benefit than the current best (maintained in a buffer *BestW*), we reset the current best appropriately. The sliding window ensures that no window of records with a higher *benefit* is missed. It can be shown that *Opt-Skip* algorithm finds the restart plan with the highest benefit.

Finally, note that even though the problem statement only bounds the number of result records cached as part of the restart plan, the working memory used by *Opt-Skip* is also O(k).

4. SOLUTION FOR THE GENERAL CASE

In this section, we extend our solution to more complex query execution plans that consist of multiple pipelines.

4.1 Execution Plans with Multiple Pipelines

A query execution plan involving blocking operators (such as sort and hash join) can be modeled as a partial order of pipelines – called its component pipelines – where each blocking operator is a root of some pipeline. For example, the execution plan in Figure 7 features two pipelines, P1 and P2. These pipelines correspond to the build side and probe side of a Hash Join respectively. In pipeline P1, the Table A is scanned and the records that satisfy the selection criteria of the Filter operator are used in the build phase of the Hash Join. The execution of P2 commences after hashing is finished. The index on Table B is scanned and records are probed into the hash table for matches.

4.2 Bounded Query Checkpointing for Multi-pipeline Plans

A multi-pipeline restart plan is obtained by replacing some subset of the component pipelines with corresponding single-pipeline restart plans. This preserves equivalence since replacing a pipeline with its restart plan preserves equivalence. For instance, in the execution plan in Figure 7, we could consider replacing pipeline P1 or P2 or both with single-pipeline restart plans.

Our goal, as with single pipeline plans is to find a restart plan such that the total state saved, counted in terms of records, is bounded and where the cost of the plan measured in terms of GetNext calls is minimized. Again, as with single pipeline plans, we introduce the notion of the *benefit* of a restart plan which is the difference in the number of GetNext calls between the initial plan and the restart plan. Thus, we are left with the online problem of maintaining the restart plan that yields the maximum benefit.

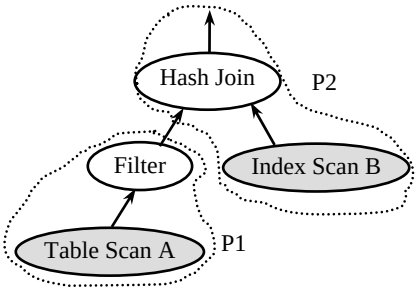

Figure 7. Execution Plan with Multiple Pipelines

4.3 Algorithms

The main difference from the single pipeline case is that for a given budget of k records, we have the option of distributing these k records among different pipelines to increase the benefit. A pipeline in an execution plan can be in one of three states – (1) completed execution, (2) currently executing or (3) not yet started. Clearly, it suffices to consider pipelines that are currently executing or have completed execution for replacement with a restart plan.

Computing the optimal distribution of *k* records in the multi-pipeline case could require excessive bookkeeping; we would need to keep track of the optimal restart plans for different values of *k*. This substantially increases the monitoring overhead during the initial run of the query. In order to keep this overhead low, we consider the following heuristic approach.

We always maintain the *BestW* window with a budget of *k* records (see Section 3.4) for the current pipeline. Whenever a pipeline finishes execution or the query is terminated, we combine this window with the windows for the previously completed pipelines so that the overall number of records to be saved is at most *k*. We outline three methods of executing this step.

Current-Pipeline: This method retains only the *BestW* window of the currently executing pipeline and ignores the windows corresponding to the previous pipelines. While simple to implement, this method could lead to poor restart plans since the benefits yielded by previously completed pipelines could be significantly higher than that yielded by the current pipeline. Our experiments in Section 5.4 confirm this intuition.

Max-Pipeline: In contrast with *Current-Pipeline*, this method takes the benefit of the previously completed pipelines into account. It only considers replacing a single pipeline with its optimal restart plan. Among all pipelines that are currently executing or have completed execution, the pipeline that yields the maximum benefit when replaced with a restart plan is chosen. This is implemented as follows. At any point, we maintain the window corresponding to the best pipeline among all pipelines that have completed execution. The *Opt-Skip* algorithm is run on the currently executing pipeline. When the current pipeline finishes execution, the benefits yielded by the windows for the current and previous pipelines are compared and the better of the two is chosen.

Merge-Pipeline: In contrast with the above methods, the Merge-Pipeline method considers distributing the space across more than one pipeline. We illustrate this method for an execution plan consisting of two pipelines. The *Opt-Skip* algorithm is used to compute the optimal restart plan for each pipeline independently. Consider the point where the second pipeline has finished executing. We now have 2 result windows cached at the roots of the two pipelines. Let these windows be (r_0,r_1 .. r_k, r_{k+1}) and (s_0,s_1 .. s_k, s_{k+1}). Since we cannot cache *2k* records, we need to eliminate some records from these windows. Suppose we wish to eliminate one record. We greedily consider eliminating each of the four corner records r_0, r_{k+1}, s_0, s_{k+1}. Among these four choices, we pick the one that brings about the least reduction in benefit. Since the budget is *k*, this process is repeated *k* times.

Sub-tree Caching: We also consider the case where the number of records returned by some node in the execution plan is less than or equal to the budget *k*. By saving all of these records, we can skip re-executing the whole sub-tree rooted at this node. We refer to this method as *sub-tree caching*. The benefit yielded by saving this set of records is set to the number of GetNext calls issued over the entire sub-tree.

5. EXPERIMENTS

In this section, we present an experimental evaluation of bounded query checkpointing. Our prototype is built by modifying the query execution engine of Microsoft SQL Server 2005. We add the primitives required for Stop-and-Restart execution outlined in Section 3.1. The clustered index scan operator is extended to accept values LB and UB and implement the skip-scan operator as described in Section 3.1. In our implementation, the saved intermediate results from the initial run are cached in memory and reused during the restart.

The main goals of the experiments are to study:

- The effect of data clustering on the benefit of bounded query checkpointing
- The utility of bounded query checkpointing for complex queries
- The overhead of bounded query checkpointing
- The different algorithms presented for the case of multi-pipeline execution plans
- The effect of the budget *k* on benefit

Databases: We use both benchmark and real data sets to evaluate our prototype. We use the TPC-H 1 GB database. The original data generator from TPC [17] does not introduce any skew in the data. We use a data generator [21] that introduces a skewed distribution for each column independently in a relation. We use a zipfian distribution with a skew factor of z = 1. Every table has a clustered index on the primary key. For our workload, we use all the queries that reference the Lineitem table; these are typically the long running queries among the TPCH suite. In order to understand how the prototype works on real data sets, we also present an evaluation using the personal edition of the Sky Server database [18] . This is a publicly available database of astronomy data.

Evaluation Metric: Queries that are terminated have to be rerun entirely in the absence of techniques suggested in this paper. We use the GetNext model of work (see Section 2.3 for details). Let T denote the total number of GetNext calls in the execution of a query Q. Let T1 denote the number of GetNext calls invoked during the initial run before the query is stopped. Let T2 be the number of GetNext calls invoked during the restart to reach the same point in execution where the query was originally stopped. Note that this point in execution is well defined for our restart plans. (T1-T2) measures the benefit of the restart plan (see Section 3.2). We define the Percentage_Work_Saved (PWS) as

PWS = (T1-T2) / T1 * 100

We instrument the query execution to write out the total number of GetNext calls seen thus far as well as the benefit value corresponding to the best restart plan until that point. After the query execution is complete, we can analyze this log to figure out what would happen if the query were terminated at any particular point in the past (for the current *k* value).

5.1 Use of GetNext Model

For the TPC-H queries used in our experiments, the leaf nodes in the query execution plan are all clustered index scans. Prior work on progress estimation indicates that the GetNext model is well-correlated with execution time for such scan-based plans [3].

We study the absolute error between the PWS values computed using execution times and the GetNext model for the following join query. The execution times are measured on a machine with 3.2 GHz CPU and 1 GB of RAM.

```
SELECT COUNT (*)
FROM   lineitem, orders
WHERE  l_orderkey = o_orderkey AND
       l_receiptdate > '1998-01-01'
```

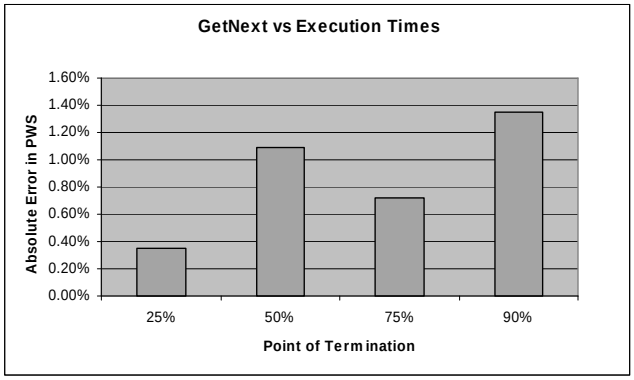

Figure 8. GetNext Model vs. Execution Times

The query plan is a hash join with the lineitem table as the build input. The budget *k* is set to 250 (chosen to be small enough for sub-tree caching to be inapplicable). Figure 8 illustrates the absolute error between the PWS computed using the GetNext model and actual execution times as a function of when the query is terminated. Note that the absolute error is uniformly low (around 1%) across different termination points of the query. We also observe similar results for different selectivity values for the predicate.

We do note that for execution plans involving operators such as index seeks, a weighted version of the GetNext model would be more appropriate.

5.2 Effect of Clustering

One of the main factors that influence the benefit yielded by the skip-scan operator is the order in which records are laid out on disk. Thus for example in Figure 4, if the records satisfying the filter predicate are evenly spaced, the benefits of bounded checkpointing would not be as significant. We study this empirically in this section.

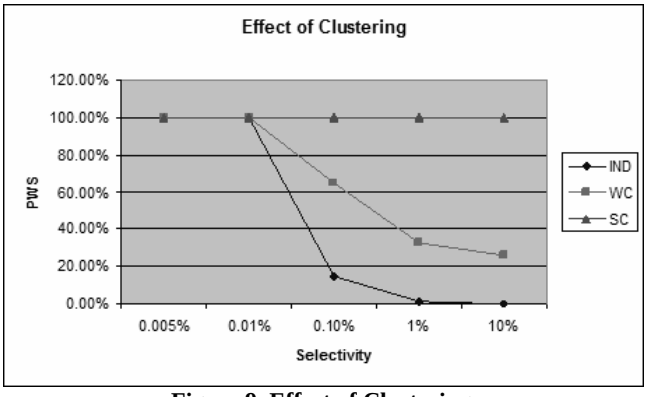

Figure 9. Effect of Clustering.

We use a join query between the lineitem and orders table which has a range predicate on the l_receiptdate column of the lineitem table. We vary the selectivity value from around 0.005% to 10% by suitably varying the predicate on the l_receiptdate column. We pick the budget *k* so that it is large enough to save all the intermediate records only when the selectivity is less than 0.01%.

For the above query, we consider three different clusterings of the lineitem table as follows.

Independent (IND): The lineitem table is clustered using the l_returnflag column. The predicate (on the l_receiptdate column) has no correlation with the clustering column. Thus, the records that satisfy the predicate are evenly spaced on disk.

Strongly Correlated (SC): The lineitem table is clustered using the l_shipdate column. Since l_shipdate and l_receiptdate are highly correlated, the records that satisfy a range predicate on the l_receiptdate are likely to be clustered together.

Weakly Correlated (WC): The lineitem table is clustered using the l_orderkey column. This represents a scenario in which the correlation between the clustering column and the predicate column falls in between the previous two cases.

Figure 9 plots the PWS values for all the 3 cases (IND, WC, SC). We terminate the query at the 50% point (that is, after 50% of the GetNext calls are over in the initial run); the results are similar for other points of termination.

For the cases where the entire set of intermediate results can be saved (the 0.05% and 0.01% cases), the PWS is maximum independent of the clustering.

As expected, a higher degree of correlation yields greater benefits. Thus, the SC case yields the maximum benefit followed by WC followed by IND.

For a given clustering of the data, increasing the selectivity of the predicate in general decreases the benefit. However, the rate at which the benefit decreases is again dependent on the specific clustering. Thus, the benefit for SC decreases much more slightly than that for WC and IND. For example, for the IND case, any set of *k*-records would have the same benefit and this would decrease linearly with increasing selectivity. In such cases, one can expect only relatively small benefits in the restart for small values of *k*.

SC represents the best-case scenario as the correlation between the l_receiptdate and the l_shipdate attributes ensures that the skip-scan operator yields maximum benefits even with increasing selectivity.

Thus, bounded checkpointing yields maximum benefit when either selectivity is low or there is a strong correlation between predicate column and the clustering column.

5.3 Evaluation on TPC-H Queries

One important factor is the overheads imposed by our techniques. As mentioned in Section 2.3, this has two components. One is the stop-response-time, which is negligible for small values of *k* (we typically set it so that all records we save can be accommodated in a few pages). The other important parameter is the monitoring overheads incurred in the initial run (when the query is not terminated). For the TPC-H workload, we observe that for most of the queries the overheads are within 3% of the original query execution times.

Figure 10 shows the PWS for the TPC-H workload when the queries are terminated at the 50% point. We set the budget *k* to be 250 (If we assume a database page size of 8KB and an average size of an intermediate record as 32 bytes, 250 records can fit in a

single page). The plots in the Figure 10 show that we can save a significant portion of the work done in the initial run by utilizing only a small amount of state. Among the 17 queries in the workload, the PWS is 40% or more for nearly half the queries in the workload.

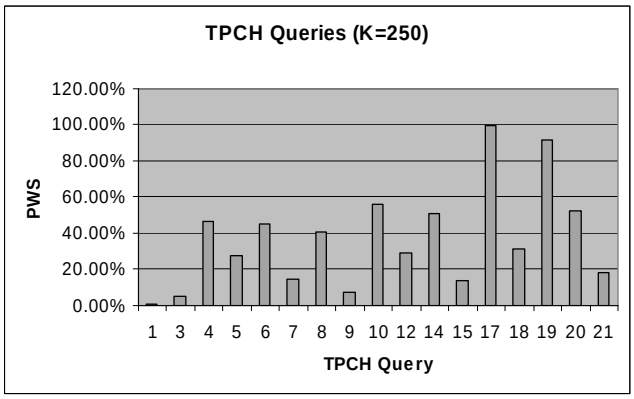

Figure 10. TPCH Queries (Termination at 50%).

Query 17 represents a case where the PWS is 100%; this is because the predicates are very selective and subtree caching is applicable. On the other hand, consider a scheme for bounded checkpointing that relies only on subtree caching, that is it saves all intermediate records, as long as the number of records does not exceed *k*. Such a scheme would only be applicable for Query 17 in the workload, while the techniques proposed in this paper yield significant benefits for a much larger number of queries.

For Queries 1 and 3, we get practically no benefit. Query 3 selects most of the records in lineitem which are evenly spaced out on disk. Thus saving just 250 records does not yield substantial benefits.

For Query 1, saving and reusing the partial sums corresponding to the aggregates would have resulted in significant benefit. In our prototype we focus on saving intermediate results and do not save any state that is internal to operators. We discuss how to incorporate partial operator state in Section 6.

5.4 Algorithm Choice for Multiple Pipelines

In Section 4.2, we presented three algorithms for the case of Multiple Pipelines (Current-Pipeline, Max-Pipeline and Merge-Pipeline). We first study the Current-Pipeline algorithm. Among the 17 queries executed for nearly half of the queries, Max-Pipeline and Merge-Pipeline yield higher benefit than Current-Pipeline, and for the remaining queries, the benefits yielded by all algorithms are equal. In fact, it can be formally shown that the Current-Pipeline algorithm yields at most the same benefit as either of the Max-Pipeline and Merge-Pipeline algorithms. Moreover, the benefits yielded by the Current-Pipeline algorithm significantly vary with the termination point. Figure 11 illustrates how the PWS varies as a function of when the query is terminated for the Current-Pipeline algorithm. For instance, if Query 10 (a join of 4 relations) is stopped at 60%, the PWS is around 25%. But if the same query were stopped at 80%, there are practically no benefits.

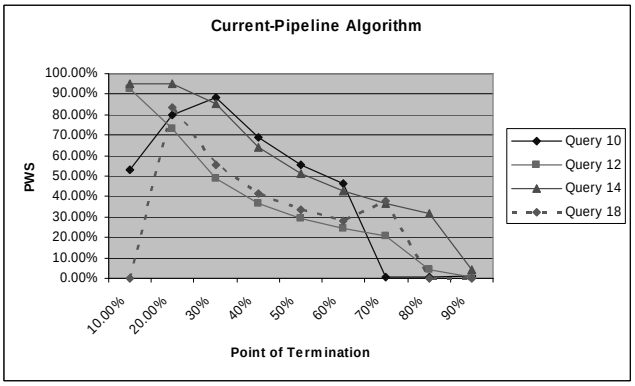

Figure 11. Current-Pipeline Approach.

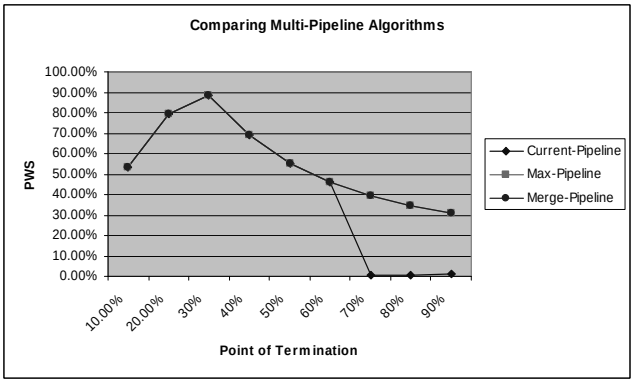

Figure 12. Multi-Pipeline Algorithms.

In Figure 12, we illustrate for Query 10 how the Max-Pipeline and the Merge-Pipeline algorithm compare to the Current-Pipeline algorithm. For this query, the best pipeline to skip is the one in which the source node is the Lineitem table. Both the Max-Pipeline and Merge-Pipeline algorithm are able to identify this unlike the Current-Pipeline algorithm. For this query, the pipelines executed after the 70% point are not good candidates for utilizing the skip-scan operator. In this case the savings obtained by using the Max-Pipeline and the Merge-Pipeline algorithm are the same.

We now compare the Max-Pipeline and Merge-Pipeline algorithms. Recall that unlike the Max-Pipeline algorithm, the Merge-Pipeline algorithm can insert the skip-scan operator at multiple pipelines. In general for TPCH queries, most of the benefit is obtained by performing a skip-scan over the lineitem table. As a result, we find that the Max-Pipeline algorithm performs as well as the Merge-Pipeline algorithm for many queries. But there are cases where utilizing the skip-scan operator at multiple pipelines is important. For instance, Figure 13 shows how the Merge-Pipeline algorithm outperforms the Max-pipeline algorithm for TPCH Query 21. Query 21 involves self-joins of the Lineitem table and the Merge-Pipeline algorithm can utilize the skip functionality on multiple scans of the Lineitem table. Notice that the savings are equal until the first pipeline finishes execution (at around 50%).

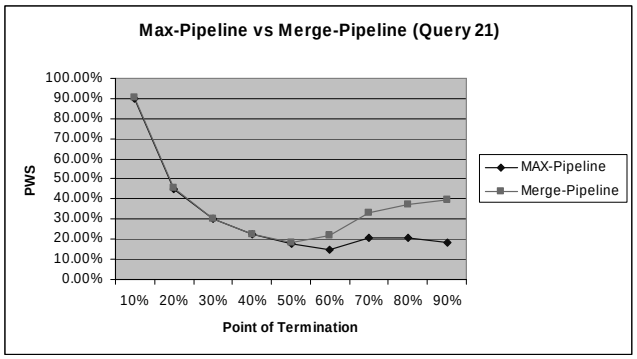

Figure 13. Max-Pipeline vs. Merge-Pipeline

5.5 Effect of Budget *k*

In Figure 14, we study how the PWS varies as a function of *k*. We show results for *k* = 1000, 5000 and 50000. We use a termination point of 50%. While it is clear that the PWS increases monotonically with *k*, the plots indicate that this is not necessarily linear. Depending on the clustering of the data, PWS can grow sub-linearly with *k*. Once *k* is large enough for subtree caching, the benefits do not necessarily increase proportionately. Further, as shown by Figure 5, the data clustering could be such that saving a small number of records clustered together yields most of the benefit after which we obtain diminishing returns as *k* increases. For nearly half the queries, we find that the PWS is almost the same for all the above values of *k*.

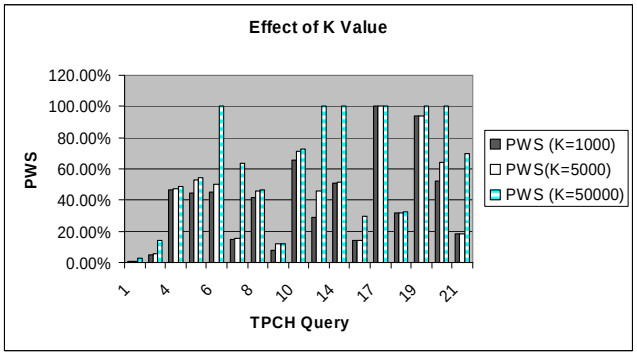

Figure 14. PWS as a function of *k*.

Of course, in general with higher values of k, the PWS increases. For instance, for the K=50000 case, we can save 100% of the work for 5 of the queries (compared to the single query in Figure 10). These results (in conjunction with those in Section 5.3) show that if we carefully choose the set of intermediate records to save, even a small amount of records can result in substantial benefits.

5.6 Evaluation on SkyServer Database

In this section, we study the effectiveness of bounded checkpointing on real data. We use the publicly available personal edition of the SkyServer database [18]. The query workload consists of 32 queries. Figure 15 shows the PWS for all the long running queries from this workload (the termination point is 50%).

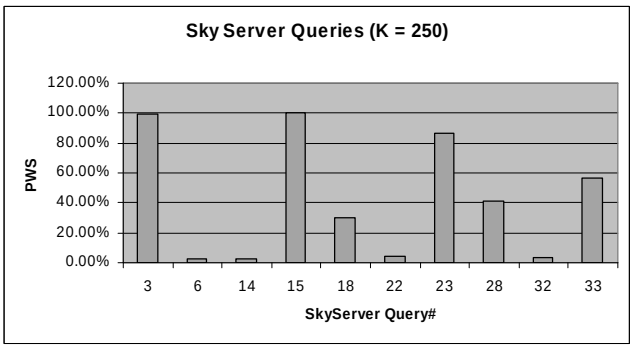

Figure 15. Evaluation on SkyServer Database

The results show that bounded checkpointing is often useful for real workloads; for nearly half of the queries the PWS is 40% or more. An important point to note is that queries 22 and 32 are single pipeline queries with no group-by or aggregations that select most of the records in the table. For such queries, bounded checkpointing is unlikely to yield significant benefit.

5.7 Summary

We summarize the experimental conclusions below

- There are many cases where bounded checkpointing is useful (selective predicates, sub-tree caching or correlation between the predicates and the clustering column). Such cases are fairly common in the real and benchmark data sets used in our experiments.
- Saving even a small amount of records can result in substantial savings in the restart run if the records are chosen carefully.
- Keeping track of previous pipelines while computing the right set of records to save is important, for instance the Merge-Pipeline algorithm is much better than the Current-Pipeline algorithm.
- The monitoring overheads imposed by our algorithms during the initial run are low. For most of the queries in the TPCH suite, the overheads are within 3% of the query execution times.

6. Discussion and Future Work

Thus far in this paper, we have introduced a space of restart plans based on the skip-scan operator. Our experiments reported in Section 5 show that we can obtain significant benefits by using the framework we have proposed. In this section, we address some open issues and while we defer a complete solution to future work, we propose possible extensions of our techniques wherever applicable.

Group by-Aggregation

One of the most common operations performed in long-running decision support queries is group-by and aggregation. The algorithms proposed in this paper handle this operation like any other operation. For example, if the number of groups output is small then subtree caching results in the entire output being saved and reused when the query is restarted. Indeed, we obtain significant benefits for the TPC-H benchmark queries as documented in Section 5.

However, we have an opportunity to do even better for group-by-aggregation in certain cases by saving partial aggregates. We explain this with an example of streaming aggregation. Consider a query that computes the expression *sum (l_extendedprice)* over the Lineitem table. During query execution, the streaming aggregation operator maintains a partial sum as a part of its internal state. We have an opportunity here to persist the partial sum when the query is stopped, and during the restart, restore the internal state of aggregate operator with the saved partial sum and skip the part of the table that contributed to the partial sum. This example generalizes to the case of group-by aggregation. Note that saving partial aggregates could potentially let us skip the *entire* work done in the pipeline in the initial run. (For example, TPCH Query 1 as discussed in Section 5.3).

Updates

Our main focus in this paper is on long-running decision support queries run on data warehouses. Data warehouses are typically maintained periodically by running a batch of updates. Therefore, it is not unreasonable to assume that the database is static as queries are run. Indeed, the discussion on equivalence of the restarted plan and the original plan presented in Section 3.4 assumes that the database is not updated between the two runs.

We now discuss how the techniques presented in this paper can be adapted to the case where the database can change as the query is executed. Whenever a query plan (involving multiple pipelines) is stopped, there is a set of pipelines which have not yet started execution. Note that if all the relations updated belong to this set and are not part of any other pipeline, the restart plan is guaranteed to be equivalent to the original plan. This observation can be used to check if the restart plan remains equivalent under updates.

We can obtain a more comprehensive way of handling updates as follows. Conceptually, we can think of the saved intermediate results as a materialized view and maintain them in the presence of updates by leveraging the extensive prior work on maintenance of materialized views [8]. We note however that unlike materialized views, the state we persist is captured using system generated rid values that are not visible at the SQL level. We would need to extend the database system to introduce the notion of *system* materialized views which are not necessarily visible in SQL.

Hash Spills

One important extension to the bounded query checkpointing problem is to enable it to handle disk spills [7]. We need additional logic to check equivalence of restart plans in the presence of hash spills. Consider an example Hash Join where the build relation is too large to fit in main memory. In this case, the join spills one or more hash partitions to disk. Assume the query execution is in the probe phase and we are computing the best-*k* records to save at the output of the join. A probe side source record for which no match is found in any of the in-memory partitions cannot be skipped since we require that all the result records produced by any skipped source record must be saved.

While a complete solution for handling spills can be complex, we could proceed by using two straightforward methods. One is to enhance the *FindSkippable* method to incorporate spills. Thus any window of records that has records that hash to a spilled partition is regarded as not skippable. An alternate approach is to disallow saving results produced by operator nodes that can potentially spill, such as hash join and hash-based group-by. Thus, for the example above, we only save the results produced by the filter below the hash join and use this to skip appropriately.

Resumption with Re-Optimization

We have assumed in the paper that the query plan used when the query is restarted is exactly the same plan used in the initial run, modulo replacing table scans with skip-scans. However, since we could potentially be skipping large portions of the base tables, we could obtain additional benefits by re-invoking the optimizer when the query is restarted. For example, suppose that we are skipping records on the probe side of a hash join. During restart, fewer records are read from the probe-side table so that it may be more efficient to perform an index nested loop join.

7. RELATED WORK

Cursors are a common interface for interacting with SQL queries and can be used to iterate over the result of a query. The execution can be paused merely by holding on to the current cursor position. An alternate approach to pausing execution is to simply suspend the query execution thread. Neither of these approaches releases the resources consumed by the query, which is the goal in this paper.

There has been prior work on pausing and resuming dataflows[19][20]. The main technical difference in our approach is that we focus on regenerating all the results (vs. generating only the remaining results) and we use a bound on the amount of state saved. We note that our techniques can be beneficial in the context of "pause and resume" implementations for pipelines whose root is a blocking operator such as a build phase of a hybrid hash join. Further, there are many scenarios where the stop-restart model of execution is more appropriate. For example, a large class of 3-tier database applications is architected to be stateless – in the event of failures (application crashes, connection or SetQueryTimeOut in ODBC), they simply start afresh.

The paradigm of saving and reusing intermediate results has been studied in various settings in prior work including view management [8], semantic caching [5][9][10], and dynamic query optimization [1][6][15]. We now distinguish our paper from each of these areas.

Recall that a stop-restart execution consists of an initial run and a restart run. We determine what to save and its corresponding restart plan during the initial run and utilize this when the query is restarted. Prior work on answering queries using views and utilizing semantic caches is applicable to the restart phase where we utilize the state saved to save computation. Our main focus in this paper is on the bounded query checkpointing problem that corresponds to the initial run of the query. This is more related to the view selection problem [1][11][12][16]. The main differences in our setting are that (1) we have a tight bound on the amount of state we are allowed to save, in contrast with a bound based on estimated size, (2) the criterion that determines what we save is the restart of the *same* query as opposed to a given workload of queries, and (3) ours is an online problem – we have to maintain a restart plan as the query execution proceeds.

Techniques for dynamic query re-optimization [1][6][15] attempt to detect sub-optimal plans during query execution and possibly re-use any intermediate results generated to re-compute the new

optimal plan. We differ in that (1) if the currently executing plan is already optimal, then query re-optimization is never invoked. However, a plan (that is optimal) can still be chosen as a victim to be terminated and restarted, (2) dynamic query re-optimization techniques do not typically constrain the number of intermediate results to save and reuse, and (3) queries are typically re-optimized by invoking the query optimizer with updated information. In contrast, we compute a restart plan online as the query executes.

8. CONCLUSIONS

Resource contention among multiple long running decision support queries could require the termination of one or more queries in current systems. However, the work done by these queries is lost even if they were close to completion. In this paper, we outlined a Stop-and-Restart style query execution that can address this problem. In particular, we looked at the bounded query checkpointing problem and presented algorithms for the same. Our experiments on a prototype system built by modifying Microsoft SQL Server 2005 indicate that our algorithms can be an interesting alternative to outright termination of queries. For many of the queries in the TPC-H suite, we could provide a significant benefit for the restart by only saving a small number of intermediate records.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments. We also thank Nicolas Bruno for helpful discussions and feedback.

9. REFERENCES

[1] Agrawal, S., Bruno, N., Chaudhuri S., and Narasayya, V. AutoAdmin: Self-Tuning Database SystemsTechnology. IEEE Data Engineering Bulletin. 2006.

[2] Babu, S., Bizarro, and P., Dewitt, D. *Proactive Re-Optimization*. Proceedings of ACM SIGMOD 2005.

[3] Chaudhuri, S., Narasayya V., and Ramamurthy, R. *Estimating Progress of Execution for SQL Queries.* Proceedings of ACM SIGMOD 2004.

[4] Chaudhuri, S., Kaushik R., and Ramamurthy, R. *When Can We Trust Progress Indicators for SQL Queries?* Proceedings of ACM SIGMOD 2005.

[5] Dar, S., Franklin, M., Jonsson, B., Srivastava, D., Tan, M. *Semantic Data Caching and Replacement.* Proceedings of VLDB 1996.

[6] Dewitt, D., and Kabra, N. *Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans.* Proceedings of ACM SIGMOD 1998.

[7] Graefe, G. *Query Evaluation Techniques for Large Databases. ACM Comput. Surv. 25(2): 73-170 (1993).*

[8] Gupta, A., Mumick, I.S., *Materialized Views: Techniques, Implementations, and Applications*. MIT Press. 1998.

[9] Haas, L., Kossmann, D., Ursu, I. *Loading a Cache with Query Results*. Proceedings of VLDB 1999.

[10] Jónsson, B. Þ., Arinbjarnar, M., Þórsson, B., Franklin, M. J., and Srivastava, D. 2006. *Performance and overhead of semantic cache management. ACM Trans. Inter. Tech.* 6, 3 (Aug. 2006)

[11] Kotidis, Y., and Roussopoulos, N. *DynaMat: A Dynamic View Management System for Data Warehouses.* Proceedings of ACM SIGMOD 1999.

[12] S.Lightstone et al. Making DB2 products self managing. Strategies and Experiences. IEEE Data Engineering Bulletin. 2006.

[13] Luo, G., Naughton, J., Ellmann, C., and Watzke, M. *Toward a Progress Indicator for Database Queries*. Proceedings of ACM SIGMOD 2004.

[14] Luo,G., Naughton, J., Ellmann, C., and Watzke, M. *Increasing the Accuracy and Coverage of SQL Progress Indicators*. ICDE 2005.

[15] Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H., and Cilimdsiz, M. *Robust Query Processing through Progressive Optimization.* Proceedings of ACM SIGMOD 2004.

[16] Roy, P., Ramamritham, K., Seshadri, S., Shenoy, P., and Sudarshan, S. *Don't Trash your Intermediate Results, Cache Them.* The Computing Research Repository cs.DB/0003005 (2000).

[17] Transaction Processing Council. *TPC-H Benchmark.* http://www.tpc.org/

[18] Sky Server Database. http://skyserver.sdss.org/

[19] B. Chandramouli, C.N.Bond, S.Babu, J. Yang. On Suspending and Resuming Dataflows. ICDE 2007.

[20] W. Labio, J.L.Wiener, H.Garcia-Molina, V.Gorelik. Efficient Resumption of Interrupted Warehouse Loads. SIGMOD 2000.

[21] S.Chaudhuri, V.Narasayya. Program for TPC-D Data Generation with Skew. ftp://ftp.research.microsoft.com/users/viveknar/tpcdskew