Operator and Query Progress Estimation in Microsoft SQL Server Live Query Statistics

Kukjin Lee[§] Arnd Christian König[§] Vivek Narasayya[§] Bolin Ding[§] Surajit Chaudhuri[§]
Brent Ellwein[§] Alexey Eksarevskiy[§] Manbeen Kohli[§] Jacob Wyant[§] Praneeta Prakash[§]
Rimma Nehme[§] Jiexing Li[∓] Jeff Naughton[∓]

[§]Microsoft Corp., Redmond
{kulee,chrisko,viveknar,bolind,surajitc,brellwei,alexek,makohli,jacobwy,pprakash,rimman}@microsoft.com

[∓]Univ. of Wisconsin, Madison
{jxli,naughton}@cs.wisc.edu

ABSTRACT

We describe the design and implementation of the new *Live Query Statistics* (LQS) feature in Microsoft SQL Server 2016. The functionality includes the display of *overall* query progress as well as progress of *individual operators* in the query execution plan. We describe the overall functionality of LQS, give usage examples and detail all areas where we had to extend the current state-of-the-art to build the complete LQS feature. Finally, we evaluate the effect these extensions have on progress estimation accuracy with a series of experiments using a large set of synthetic and real workloads.

1. INTRODUCTION

Accurate online estimates of the overall progress of execution of a SQL query is valuable to database administrators (DBAs), application developers and end users of applications. For example, a DBA (or an automated agent) can use query progress information to help decide whether a long-running, resource intensive query should be terminated or allowed to run to completion. There has been extensive work in SQL query progress estimation, e.g., [6, 18, 23, 24].

In addition to overall query progress information, it can also be valuable to provide progress estimates for *individual operators* in a query execution plan [12]. Such an operator-level progress indicator can aid in live troubleshooting of performance issues due to a poorly chosen execution plan: similarly to how accurate progress estimation allows database administrators to identify long-running queries early in their execution, having accurate progress estimates for individual operators allows DBAs to rapidly identify issues with operators that require significantly more time or resources than expected. For example, a database administrator might observe a nested loop operator that is not only executing for a significant amount of time, but, according to the progress estimate, has only completed a small fraction of its work. Based on this, she may then compare the number of rows seen so far on the outer side of the join and discover that these are already much larger than the optimizer estimate for the total number of outer rows, indicating an cardinality estimation problem. This may then trigger various remedial actions on the side of the DBA, such as creating additional statistics, the use of plan hints or a reformulation of the query. The same determination could obviously also be made using the operator duration information only, obtained *after* execution; however, having online operator-level progress information enables a DBA to identify potential issues much more quickly and easily.

In this paper we describe the design and implementation of operator and query level progress estimation included as part of Microsoft SQL Server 2016, available via the new Live Query Statistics (LQS) feature that is part of SQL Server Management Studio (SSMS). This feature enables the scenarios discussed above by providing estimates of both query and operator level progress. These progress estimates as well as the row counts are exposed to users through a query plan visualization. LQS can work with both Microsoft SQL Server 2014 as well as Microsoft SQL Server 2016.

1.1 Overview and Organisation

In the following, we will describe the new Live Query Statistics feature in detail; first, in Section 2, we will give an overview of the LQS functionality, including a description of the different components working together, and show examples of the output surfaced to the user. Subsequently, in Section 3, we will formally introduce the relevant technical background and then describe the particular challenges we faced when building the LQS feature. In particular, we will identify all aspects in which we needed to extend the current state-of-the-art to arrive at the desired functionality and level of accuracy. We will then go through the details of how we addressed each of these challenges in Section 4 and experimentally evaluate the effect of the resulting techniques in Section 5. Finally, we briefly discuss future work in Section 7.

2. OVERVIEW OF LQS

In this section, we will give an overview of the *Live Query Statistics* functionality as well as the different instrumented components. The progress estimator itself is a client-side component, which consumes all required counters for currently executing queries exposed via *dynamic management views* (DMV) on the server side (see Figure 1). The progress estimate for both the entire query as well as each individual operator is then computed in the client and displayed visually together with the query plan.

SIGMOD '16, June 26–July 1, 2016, San Francisco, CA, USA.

ACM 978-1-4503-3531-7/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2882903.2903728

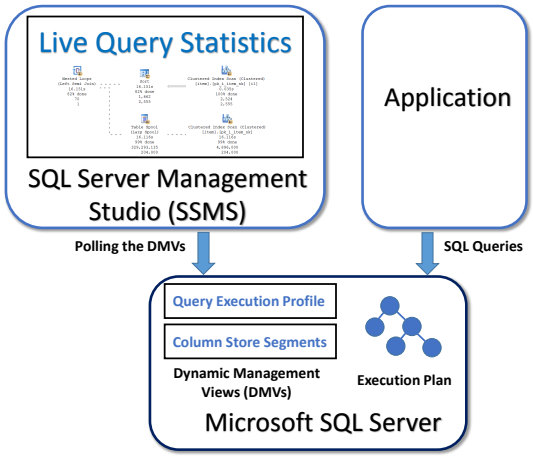

Figure 1: Architecture Overview.

2.1 Server Side DMVs

The counters required for progress estimation are collected within the Microsoft SQL Server engine process; most of them are exposed by the dynamic management view *sys.dm_exec_query_profiles* introduced first in Microsoft SQL Server 2014[1]. This view returns counters for each operator in a *currently executing query* (at the granularity of individual threads), such as the estimated and actual number of rows, the total elapsed time and CPU time for an operator, the number of physical reads, etc. Each counter is dynamically updated while the query executes. LQS supports the display of progress estimates for multiple, concurrently executing queries, each of them being given their own dedicated window.

2.2 Progress Estimation in SSMS

The progress estimation itself is implemented as a client-side module which is part of the SQL Server Management Studio. This model queries the DMV *sys.dm_exec_query_profiles* every 500ms to obtain up-to-date information on each executing operator/thread and, in addition, uses information extracted from a query's *showplan* (such as estimated cardinalities as well as CPU and I/O cost estimates) as well as additional system meta-data (such as e.g., the DMV *sys.column_store_segments*). The fact that progress estimation is client code (with the required information exposed by the server) has the advantage that the progress estimation module can evolve independently of the server code. Further, it allows for the development of other first-party or third-party client side tools that use the same data sources.

The main disadvantage lies in the fact that some information that is potentially useful for progress estimation (such as more detailed information on the internal state of *Sort* and *Hash* operators) is not exposed via *sys.dm_exec_query_profiles* at this time. Because the progress estimation is client code, it has no visibility into some aspects of operator state. We will touch on some of these limitations in Section 7.

2.3 Visualization

In SQL Server Management Studio, progress estimates are shown combined with *query showplan* output, showing an estimate of progress of the overall query (i.e., percentage completion) as well as an progress estimate for each operator as well. An example can be seen in Figure 2. In this example, the query is TPC-H Query 1, and the *showplan* for this query is shown in the bottom window. Below each operator, its progress is displayed – in this example, the Columnstore *Index Scan* and *Compute Scalar* operators have executed for 42.58 seconds and are at 5% progress, whereas all operators further up the plan have not started execution yet. The overall query progress is displayed on the top left-hand corner of the *showplan* window (3% in this example).

2.3.1 Visualization and Usage Examples

The LQS interface animates the links between all currently executing operators, thereby making it easy to identify which pipelines in the query plan are active at any moment. For example, in Figure 3 we can see that the upper two pipelines have already completed execution; here, the arrows between two operators are solid and their thickness corresponds to the number of tuples having flown between them. In contrast, the lower pipeline has dotted, animated arrows between the operators, meaning that it is still executing and – since all operators in the pipeline are still at 0% progress after 40 seconds of execution – likely long-running. Especially for complex query plans, this functionality allows DBAs to identify potential bottlenecks and opportunities for tuning very quickly.

In Figure 4 we demonstrate a different example scenario for LQS: here, in addition to the output we described earlier in the context of Figure 2, we also show the estimated cardinality and the total number of rows output so far for each operator in the plan[2] for illustration. As we can see, while the upper *Clustered Index Scan* operator has already finished execution (we can see the progress of the operator being reported as 100%), the *Table Spool* and lower *Clustered Index Scan* operator are still executing and show very significant errors in estimated cardinality; this cardinality information allows the DBA to identify nodes and the corresponding predicates with significant estimation errors which potentially may contribute to sub-optimal plans.

As we can see in Figure 4, the progress estimates at the *Table Spool* and lower *Clustered Index Scan* operators themselves are not accurate, which in turn leads to over-estimation of the fraction of work done at these nodes. We chose this example to illustrate that even when we do not have accurate estimates of node cardinality and e.g., overestimate the work done by an operator, LQS can still guide a database administrator to problematic nodes in the execution plan: in this example, the progress estimates at these nodes initially go up to 99% and then stay at that estimate for a significant time time. This in itself is enough to alert a DBA to the inaccuracy and potentially trigger an investigation of the nodes in question.

3. CHALLENGES

In this section we will describe the challenges encountered during the implementation of the LQS functionality. In particular, we will describe where the current state-of-the-art in research either needed to be extended or did not yield sufficient accuracy for some of the real workloads we used to evaluate the LQS system. To give this overview, we will first introduce some necessary background on progress estimation.

3.1 Background

3.1.1 Operators and Pipelines

Current progress estimation techniques [6,7,12,15,21–24] model a running query Q using a tree of physical operators and base their estimates of a query's progress on the number of tuples or bytes

[1]See `https://msdn.microsoft.com/en-us/library/-dn223301.aspx` for details.

[2]In the current interface, this information is is available by hovering the mouse over the operators in the *showplan*.

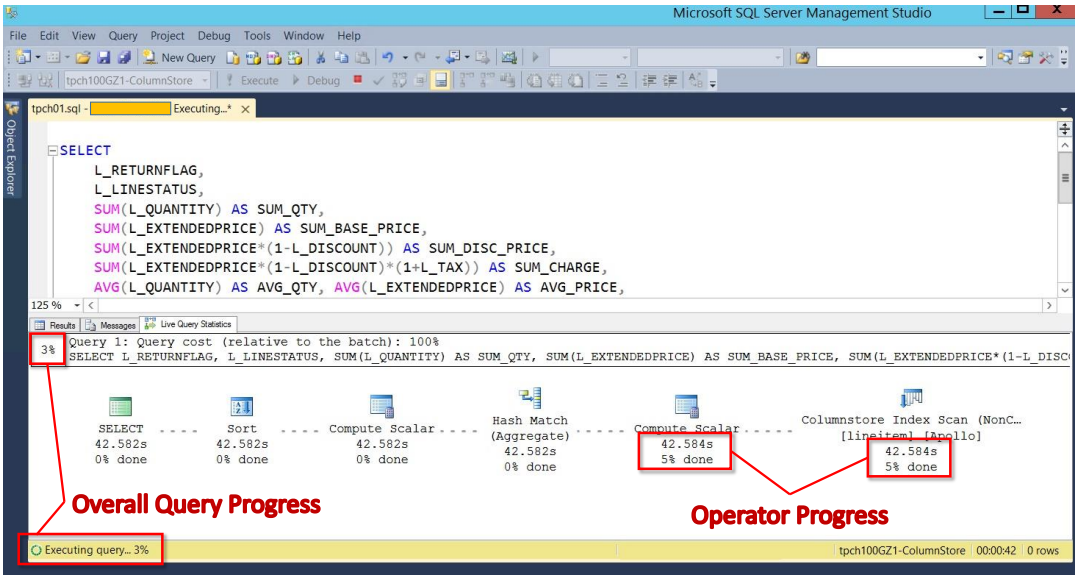

Figure 2: Query and Operator Progress in Live Query Statistics

processed by (a subset of) these operators at various points in time. We use $Nodes(\mathcal{Q})$ to enumerate the operators in the execution plan. To capture the notion of the sets of operators in a query plan which execute concurrently, prior work defined *Pipelines* or *Segments* (defined in [7], and [22], respectively), which correspond to maximal subtrees of concurrently executing operators in a (physical) operator tree.

For each pipeline (or segment), the nodes that are the sources of tuples operated upon by the remaining nodes (i.e., typically all leaf nodes of the pipeline, with the exception of the inner subtree of nested loop operators) are referred to as the *driver nodes* (or *dominant input*) of the pipeline; we use the notation $DriverNodes(\mathcal{Q}) \subseteq Nodes(\mathcal{Q})$ to denote these. An example operator tree with 3 pipelines is shown in Figure 5; the shaded nodes correspond to driver nodes. The notion of segments was further refined in [23].

3.1.2 Model of Work Done

Operators in a query execution plan of a modern RDBMS are typically implemented using a demand driven iterator model [11], where each physical operator in the execution plan exports a standard interface for query processing (including Open(), Close() and GetNext()). For the purposes of progress estimation, we follow the $GetNext$ model of work done by an operator, introduced in [7]. In the $GetNext$ model, at any point during the execution of the query the true progress of an *operator* o is defined as:

$$Prog(o) = \frac{k}{N} \tag{1}$$

where k is the number of rows output by the operator thus far, and N is the total number of rows that will be output by the operator when query execution is complete. Similarly, the true progress of a *query* Q in the $GetNext$ model is:

$$Prog(Q) = \frac{\sum_{i \in Nodes(\mathcal{Q})} w_i k_i}{\sum_{i \in Nodes(\mathcal{Q})} w_i N_i} \tag{2}$$

The summation is taken over all operators in the query execution plan. w_i is a *weight* associated with operator i that can be used to capture the fact that different operators perform different amount of work *per tuple* output relative to other operators.

While the k_i values can be measured precisely and with negligible overhead at query time, the N_i values are typically derived using the estimates of query optimizer, meaning that the accuracy of progress estimation is closely tied to the accuracy of cardinality estimation itself [15], which is know to erroneous in many cases. One of the ways this challenge has been alleviated has been by only considering driver nodes in the summation in equation 2 [7]. Because the cardinalities of driver nodes are typically known exactly when their pipeline starts execution (although there are notable exceptions which we will discuss in subsequent sections), this reduces the error in progress estimation for executing pipelines significantly. The main requirement for this approach to work is that the progress at the driver nodes closely matches the progress over all nodes of the pipeline.

3.2 Progress Estimation for Individual Operators

While previous work in progress estimation only concentrated on estimating the overall progress of queries (with the exception of [12]), Live Query Statistics also surfaces progress estimates for each individual operator. Providing operator-level progress estimates is conceptually similar to the *GetNext* based scheme described above (see Equation 1), when limited to single nodes, but creates a number of additional challenges, which we will describe below.

3.3 Challenges and Contributions

In order to enable the LQS functionality, there were a number of challenges we had to address. First, a key challenge for any progress estimation technique that ties their measure of progress to the number of rows output or processed by each operator is to accurately estimate the total number of rows to be output/processed, which is closely tied to the known hard problem of cardinality estimation. This challenge is made more difficult in our case by the fact that LQS provides operator-level progress estimates and hence requires accurate cardinality estimates for *all* operators in a plan; the

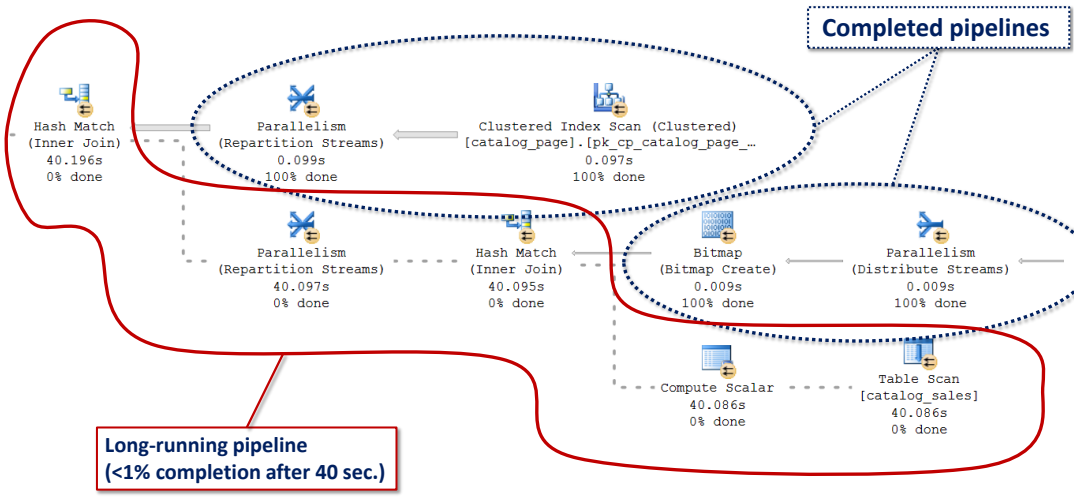

Figure 3: Live Query Statistics in Action

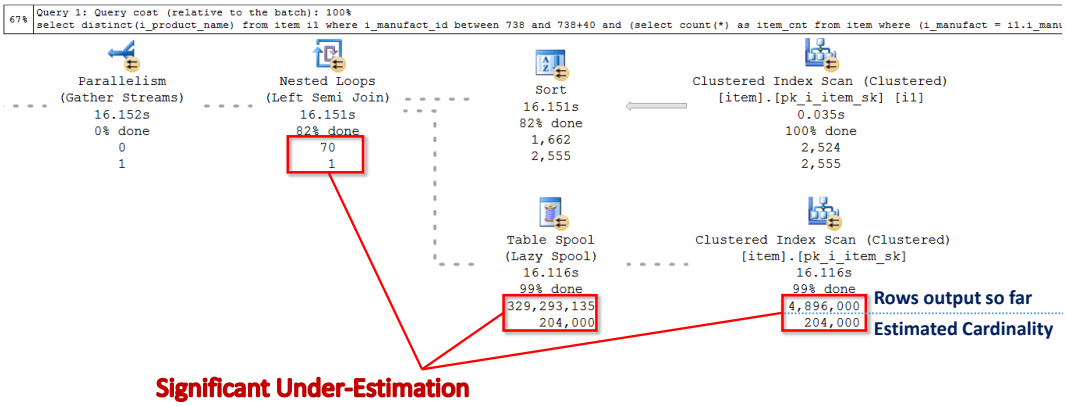

Figure 4: Live Query Statistics in Action

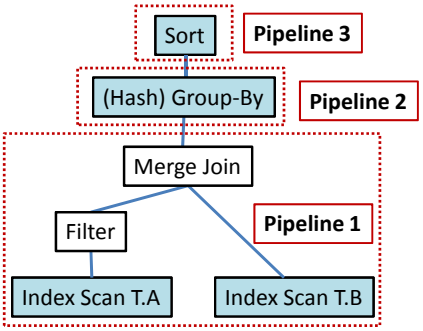

Figure 5: Example execution plan with multiple pipelines.

"trick" described above of only considering driver nodes can not be applied here, unlike for progress estimation of the entire query (or pipeline).

While the challenge of inaccurate cardinality estimates cannot be overcome generally, there is one unique aspect of progress estimation that can be leveraged here: unlike the cardinality estimation inside the query optimizer [5], progress estimation has the opportunity to observe execution characteristics of operators *in flight*, and use this feedback to dynamically *refine* cardinality estimates for both the executing operators and operators further up in the query plan. Earlier approaches to progress estimation [6, 15, 23] have used this to propose techniques that dynamically refine cardinality estimates as the query executes. For LQS, we propose a cardinality refinement approach of the same type, which uses a different model (described in Section 4.1) and in practice demonstrates fast convergence to the true cardinality. We combine this technique with the (worst-base) bounding techniques proposed in [7] based on algebraic properties of the operators (which we describe in Section 4.2). The combination of both techniques can help us mitigate many cases where there are egregious errors in the initial optimizer cardinality estimates.

Second, for accurate estimates of *operator* level progress, we needed to develop new techniques for modeling blocking operators (e.g. *Hash*, *Sort*), whose progress is in general insufficiently characterized by the number of tuples output by the operator itself. Here, the unmodified *GetNext* model as described in Section 3.1.2 is not sufficient to give accurate progress estimates, even when cardinalities are known. In addition, similar but not identical challenges are posed by *semi-blocking* operators (e.g., *Exchange*, *Nested Loops*) that can buffer inputs and break key assumptions

made by earlier progress estimation techniques. We describe the techniques used to address these challenges in Sections 4.4 and 4.5.

Third, due to the often vastly different (per-row) resource requirements of different operators, we need to determine an appropriate weight for each operator in a query to obtain accurate overall query progress. For this challenge, we adopt a variant of the technique of [18], which leverages per-tuple CPU and I/O cost estimates by the optimizer and models the overlap between CPU and I/O as the basis for automatically determining an appropriate weight for an operator (or pipeline of operators).

Finally, we need to handle a number of additional cases not addressed in prior work but common in practice, including: (a) Scan operators containing filters evaluated at the storage engine level (e.g., *Bitmap* filters) whose selectivity may initially be estimated incorrectly, (b) operators such as *Scans* of *Columnstore indexes* [16, 17] that execute in *batch mode* rather than one row at a time. We will describe how we address each of these challenges in the next section.

4. ADDRESSING THE TECHNICAL CHALLENGES

In Section 3.3, we have identified a number of different challenges to accurate progress estimation as well as some scenarios not addressed by prior work; in this section, we will describe different techniques used to address these and improve the accuracy of progress estimation.

4.1 Cardinality Refinement

As described in Section 3.1.2, the N_i terms in the progress estimator (see Equation 2) are based on optimizer estimates, which may be very inaccurate for the long-running and often complex queries that progress estimation is targeted at. The use of *driver nodes* discussed in Section 3.1.1 alleviates this concern only partially; this approach assumes that the overall progress in a pipeline is proportional to the progress of the driver nodes, which can be violated in practice (see [15], Section 4.4.1, for a more detailed discussion). Moreover, providing operator-level progress estimates requires accurate estimates of the N_i terms on all nodes, not just the driver nodes.

While there exist large numbers of techniques to improve cardinality estimates by leveraging feedback from other executing queries in the research literature (such as [4, 8, 14, 20, 27, 28]), many of which are applicable in the context of progress estimation, our current technique only leverages the data structures and statistics already available in the target database engines; we leave the integration of novel statistical summaries as future work.

Instead, we leverage the observation that, unlike query optimization, progress estimation can leverage improved N_i estimates obtained *after* a query has started execution. The basic technique we use is to refine the estimate of N_i at runtime by treating the input tuples seen at a node i as a random sample of all tuples processed by this operator (similarly, to e.g., [13] for $COUNT$ aggregate queries) and scale the corresponding k_i up by the (inverse of the) fraction α of the tuples consumed at the driver node(s) of the corresponding pipeline:

$$\alpha = \frac{\sum_{i \in \text{DriverNodes}} k_i}{\sum_{i \in \text{DriverNodes}} N_i} \tag{3}$$

Here, we use only the driver nodes to estimate the scale-up factor, as for them the N_i are typically known with high accuracy, making the estimate of the total cardinality $N_i = k_i/\alpha$. The refinement of the total cardinality N_i at node i, i.e., $N_i = k_i/\alpha$, can be interpreted as *population estimation* using sampling without replacement. The intuition is that, each row output by a driver node (by a *GetNext* call) may trigger the output of zero, one, or more rows at node i. So assuming future drive-node rows trigger the output of rows at node i at *the same rate*, the total number of rows output by node i becomes k_i/α.

Of course, rows are output at node i at varying rates during the whole process and thus we need to estimate the *average rate*. Formally, let

$$TotalRows_{\text{DN}} = \sum_{i \in \text{DriverNodes}} N_i$$

be the total number of rows output by driver nodes, and

$$SeenRows_{\text{DN}} = \sum_{i \in \text{DriverNodes}} k_i$$

be the number of rows output by them so far. Let N_i^{true} be the true cardinality at node i. Our estimation N_i thus can be rewritten as

$$N_i = \frac{k_i}{\frac{SeenRows_{\text{DN}}}{TotalRows_{\text{DN}}}} = \frac{k_i}{SeenRows_{\text{DN}}} \cdot TotalRows_{\text{DN}}.$$

The true cardinality N_i^{true} can be rewritten similarly to be

$$N_i^{true} = \frac{N_i^{true}}{TotalRows_{\text{DN}}} \cdot TotalRows_{\text{DN}},$$

where $N_i^{true}/TotalRows_{\text{DN}}$ is the average rate of rows being output at node i. We essentially estimate this true average rate as $k_i/SeenRows_{\text{DN}}$, which decides how much error there is in our estimation of N_i relative to N_i^{true}.

If rows from driver nodes are randomly permuted, $k_i/SeenRows_{\text{DN}}$ is a reasonably accurate estimation of $N_i^{true}/TotalRows_{\text{DN}}$ with the estimation error bounded by the *Central Limit Theorem* or *Hoeffding inequality*. The order of these rows, unfortunately, depends on the physical layout of database tables which is fixed in advance and cannot be assumed to be totally random. Still, refining cardinalities in this manner is still a valid strategy for the vast majority of possible orderings: we have been able to show that among all possible orders of these rows, in the majority of them, $k_i/SeenRows_{\text{DN}}$ is very close to $N_i^{true}/TotalRows_{\text{DN}}$.

Formally, an order of rows from driver nodes is said to be a *bad order* iff $\frac{|N_i - N_i^{true}|}{N_i^{true}} > \epsilon$. Among all possible orders of rows being output from driver nodes, the portion of bad orders is proportional to $\frac{1}{\epsilon^2 \cdot SeenRows_{\text{DN}}}$ (where the exact equation depends on a number of additional, data-dependent parameters), which decreases quickly as we get more and more rows from driver nodes (i.e., as $SeenRows_{\text{DN}}$ increases).

Obviously, as also pointed out in [12], extrapolating the N_i from observations in this manner requires initially observing sufficiently many tuples at node i as well to allow us to estimate the operator selectivity accurately. As a result, we only invoke the cardinality refinement if the following guard conditions are met: First, similar to [12], we require a minimum number of tuples observed for all inputs to any operator. Second, for any filter or join operator, we only trigger refinement if we have observed *both* tuples satisfying the join/filter condition as well as tuples not satisfying it.

One thing to note is that a different techniques for online refinement of cardinality estimates in progress estimates have been proposed previously [22]. Unlike this approach, we do not use linear interpolation between the initial optimizer estimate and the scaled-up estimate, as we found these estimates to converge very slowly for highly erroneous initial estimates.

Finally, for any nested loop iterations, we make an additional modification to the technique, refining the cardinality estimates on the inner side of the nested loop by scaling the average number of $GetNext$ calls per input tuple by the corresponding $1/\alpha$ for the outer side; this is merely reflecting the fact that for nested loops, the outer side of the loop governs how often the inner side is invoked (as opposed to the child node). In case of multiple nested loops within a pipeline we have to apply this logic multiple times, from the outer to the inner nodes.

4.2 Cardinality Bounding

In addition to online cardinality refinement, we use the techniques described in [7] to maintain *worst-case* lower and upper bounds on the number of $GetNext$ calls possible for each operator, based on the number of tuples seen so far at each operator and the size of the input(s). If the value of any N_i ever falls outside of these bounds, it is set to the nearest boundary value. We describe the computation of the cardinality bounds on a per-operator basis in Appendix A.

Since these bounds are derived based on the algebraic properties of the operators alone, they are conservative, and therefore tend to only kick in for operators in the later pipelines in a query plan; for these operators, the initial cardinality estimation often requires computing selectivities for multiple joins and filter predicates, meaning that the cardinality estimates often have significant errors. In these cases, cardinality bounding can reduce these errors significantly: for example, after a first pipeline is complete, we know the exact cardinality of the input to the leaf node(s) of the second pipeline. In such cases, the resulting upper bound on the corresponding N_i for any node in the second pipeline can be significantly lower than the initial optimizer estimate of these N_i, in turn improving the cardinalities used during progress estimation. Similarly, these bounding techniques can improve estimates in case of cardinality underestimation as well. As we will show in the experimental evaluation (see Section 5.1), cardinality bounding results in noticeable increase in overall progress estimation accuracy.

4.3 Predicates evaluated in the Storage Engine

A special case for progress estimation are predicates that are pushed down to the storage engine. One important example of this are *bitmap filters*, which are common in data warehousing workloads. Bitmap filters are a type of semi-join reduction in which rows on the inner side of the join that cannot join with rows on the outer side, are eliminated prior to the join operator. In a typical case, the bitmap filter is created when the build input of a *Hash Join* is executed, and the bitmap filter is evaluated when scanning the probe side input. An example of a bitmap filter that has been pushed into the scan is shown in Figure 6.

There are many other examples of such filters, such as complex combinations of conjunctions and disjunctions, predicates on out-of-model scalar functions, etc., all of which may be pushed to the storage engine.

Because these predicates filter out rows as part of the initial data access, the assumption that we base driver nodes on, namely, that we have accurate estimates of the number of rows output by a driver node, is not valid any more. In fact, the bitmap filters and other complex predicates of this type often have very large estimation errors. As a result, using these driver nodes to compute query progress or as part of cardinality refinement for other nodes becomes problematic.

Therefore, we use a different technique when estimating the progress at nodes of this type: as long as only predicates are pushed to the storage engine that cannot be supported by an index, we know that

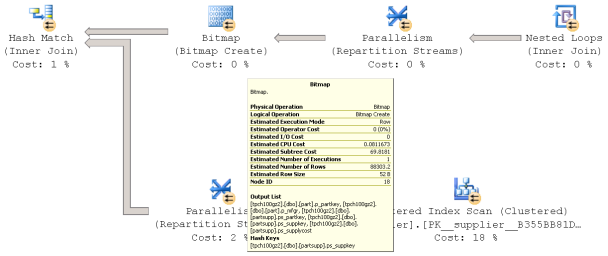

Figure 6: Plan with Bitmap Filter Evaluation in Index Scan.

the entirety of the table has to be accessed by the *Scan* operator (even if many rows may be filtered as part of the scan); hence, we instead compute the number of logical I/O operations required to scan the underlying table and based the current measure of progress at the *Scan* node in question on the fraction of these I/O requests issued at this point. This approach has turned out to be highly accurate and a significant improvement over the initial approach.

4.4 Semi-blocking Operators

Some operators such as *Sort* and *Hash* are stop-and-go, i.e., they are blocking. Other operators are pipelined in their execution, however, they buffer their output, which causes them to behave similar to blocking operators for subsets of their output. We refer to such operators as *semi-blocking*. Examples of such semi-blocking operators include *Nested Loops* (when the inner operator is an Index Seek) and *Parallelism* (aka. *Exchange*).

To illustrate this issue, consider the query pipelines shown in Figure 7: without any buffering, we would expect the number of *GetNext* calls in the *Parallelism* operator and its *Nested Loop* child to be identical. However, this is not the case in practice: in Figure 8 we plot the number of *GetNext* calls for both operators over time – as we can see, the *Parallelism* operator 'lags' behind its child noticeably. Note that while the curves follow each other closely on the time-scale, because of the steep trajectory of the curve, the difference in the number of *GetNext* calls between the operators can be very significant; for example, for the two measurements highlighted in Figure 8, the ratios between the *GetNext* calls of the *Nested Loop* and the *Parallelism* operator are about 88x and 12x, respectively. This, in turn, can significantly hurt the accuracy of progress estimation for two main reasons.

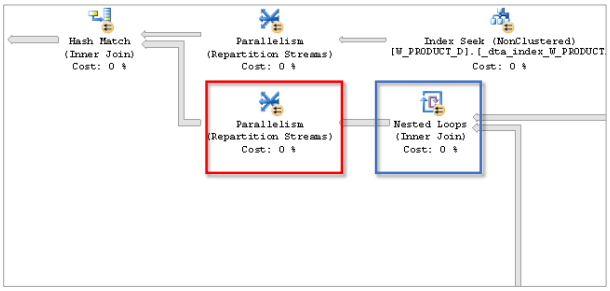

Figure 7: The *Parallelism* operator in this plan progresses significantly slower than its child node (*Nested Loop*)

First, they break the central assumption underlying the use of *driver nodes*, namely that the progress measured using driver nodes only corresponds to the progress over all nodes in a pipeline. To illustrate this, consider the case of a pipeline consisting of a single index nested loop join, with the scan of the outer side being the sole driver node. In this scenario, the buffering can lead to the

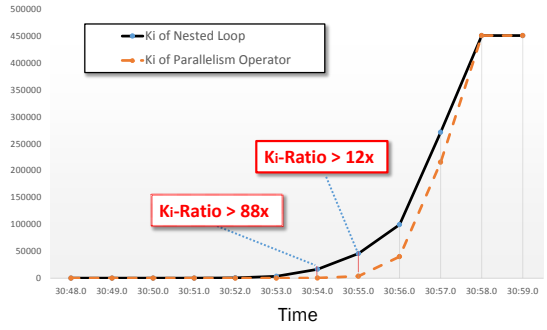

Figure 8: The difference in the *GetNext* calls between the two nodes from Figure 7 over time

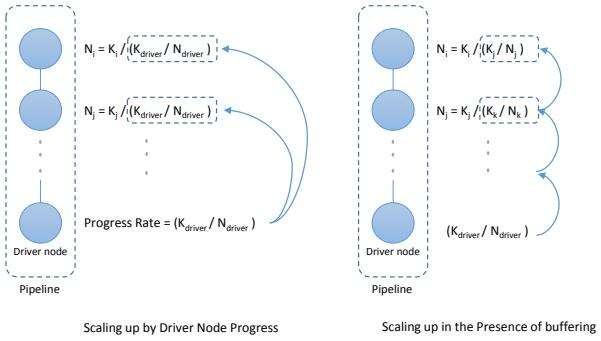

Figure 9: Scaling up N_i estimates using driver nodes vs. immediate child nodes.

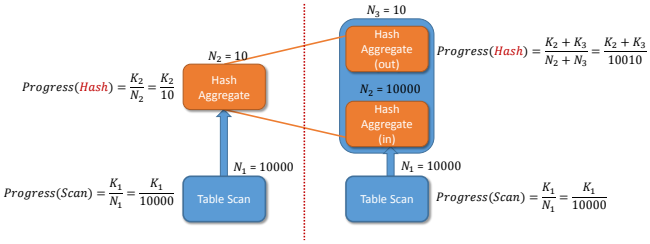

Figure 10: Modeling Hash Aggregation Progress in Two Phases.

scenario where all tuples from the outer side have been consumed and buffered before any tuples on the inner side were accessed, meaning that the progress at the driver node is 100%, yet the query may still be far from completion. This is a scenario we observe often in practice.

Second, when significant buffering occurs, the method described in Section 4.1 of refining the N_i estimates by scaling the corresponding k_i by the (inverse) driver-node progress is not accurate any more; especially when multiple semi-blocking operators occur in sequence in a query pipeline, this will lead to significant overestimation of the N_i terms, as the 'correct' scale-factor becomes smaller and smaller with every semi-blocking operator between the node whose cardinality we are trying to refine and the driver nodes of the corresponding pipeline.

To address these issues, we modify the techniques described so far for pipelines containing semi-blocking operators as follows:
(1) For progress estimation of *Nested Loop* joins (which buffer tuples coming from the outer side), we treat the inner side of the join as a driver node as well. Similar modifications were proposed in [15] and were shown to yield significant improvements across a wide range of workloads there.
(2) For the purpose of improving initial estimates of the N_i counts using the cardinality refinement techniques of Section 4.1, we scale-up the k_i counts of a node that is separated from the leaves of its pipeline by at least one semi-blocking operator, using the progress at its *immediate child*. This is in contrast to using the progress of the driver node to scale-up the k_i counts. This difference in scale-up logic is illustrated in Figure 9. In case of nodes with multiple children (e.g., *Merge Joins*) we use both children's progress to compute the corresponding α.
(3) Finally, we adjust the scale-up factor for tuples on the inner side of a nested loop join to adjust for the fraction of tuples buffered on the outer side of the operator.

4.5 Improvements for Blocking Operators

One challenge introduced by operator-level progress is the accurate modelling of progress for blocking operators. For many blocking operators, the fraction of work done by the operator itself is very poorly characterized by the fraction of tuples output by the operator. For example, in case of *Hash Aggregates*, once tuples have started to be output, typically most of the processing and I/O associated with the operator have been completed already. When considering query-level progress, this issue is often, but not always, masked by the fact that the I/O and processing associated with tuples being input to a blocking operator is modelled as part of its child's operator(s). However, for operator-level progress, this does not apply any more.

Earlier work [12] has dealt with this scenario by introducing two new terms: "blocking time", which specifies the amount of time required before an operator outputs tuples, and "blocking progress", which specifies the fraction of total work done before tuples are output. The approach we chose in this work is different: because the relative per-tuple processing overhead for input and output tuples can be significantly different and change as a function of the width of output rows, complexity and number of aggregate expressions and grouping columns, etc., we model operators of this type (such as *Sort* and *Hash*) as two distinct phases. The first phase progresses in proportion to the fraction of input tuples consumed, whereas the second phase progresses in proportion to the fraction of tuples output by the operator.

We show the change compared to earlier approaches in Figure 10. On the left side we show the earlier progress model for a *Hash Aggregate* operator which inputs 10K tuples and outputs 10. In this model, the progress for this operator is 0 until tuples are output. In the new model, shown on the right side, the input and output phases of the *Hash Aggregate* are modelled separately, with the input executing synchronously with the child *Scan* operator and the output executing subsequently. To illustrate the effect of this change in practice, we plot the example progress for a *Hash Aggregate* operator (used in the execution of TPC-DS Query 13) in Figure 11 over the time the operator is active; in the figure, the dotted line represents a perfect estimate. Whereas the original model (using the output N_i only) reports no progress throughout nearly the entire time (and only changes when output is produced at the very end of execution), the new progress model (based on both input and output tuples) shows the operator making progress before any output is produced and correlates much better with time.

Note that the new model still can be an over-simplification, for example, for large sorts with multiple merge steps even more intri-

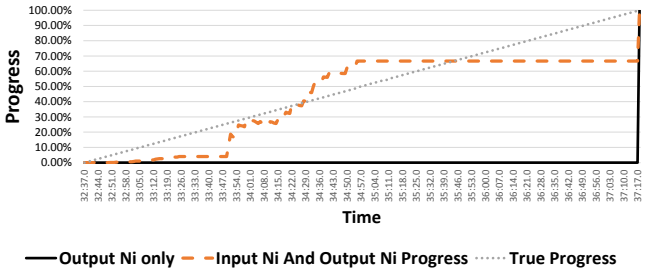

Figure 11: The effect of the refined model for Hash Aggregation

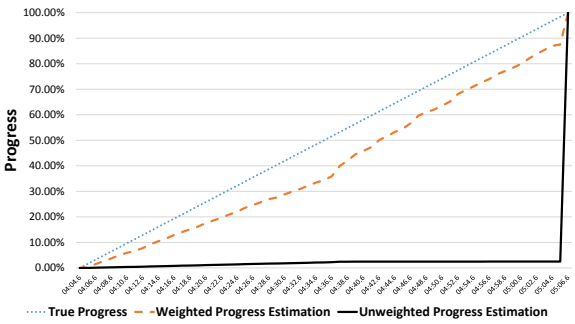

Figure 12: Progress for TPCDS Q21 with and w/o operator weights

cate models may be needed; however, it performs well in practice across a variety of workloads and significantly improves upon our earlier models, as we demonstrate in the experimental evaluation.

4.6 Selecting the Operator Weights

While the accuracy of single-operator progress mainly depends on accurate estimation of the N_i (and – for blocking operators – how we "break" them into pipelines), the accuracy of the overall query progress requires us to assess the relative "speeds" at which different pipelines iterate over tuples; these speeds in turn depend on the corresponding row sizes and the amount of processing done at each individual operator in a pipeline.

We model these differences in speeds by changing the corresponding w_i terms in equation 2 accordingly. For this, we adopt the framework first introduced in [18]; here, the query workload is divided up into *speed-independent* pipelines, which are defined as groups of connected operators that execute concurrently and process tuples at a speed independent of the speeds of other operators (in other pipelines). The relative weights for each such pipeline are then set using estimates of I/O and CPU cost of the corresponding operators by the optimizer and the notion of 'overlapping' work: we make the (simplifying) assumption that I/O and CPU cost within a single operator are overlapping, meaning that only their maximum is used to weigh a pipeline.

In addition, the overall duration of a query depends only on the duration of its "longest" path (i.e., the path of speed-independent pipelines from the top operator in a query to the leaf level which requires the longest time among all such paths). As a result, we maintain this path (based on optimizer cost estimates of I/O and CPU cost per tuple and refined N_i counts) among all pipelines and use only the set of nodes along it to compute the overall query progress.

Because the weights we use are derived from optimizer cost estimates, they do not model transient effects not considered by it (e.g., effects of caching due to the buffer pool).

To illustrate the difference that this weighting scheme can make on progress estimation, we plotted the progress estimates over time for query 21 from the TPC-DS benchmark in Figure 12. As we can see in Figure 12, the un-weighted estimator severely underestimates query progress until the very end of query execution. In contrast, the weighted operator progress correlates much better with time. Examining the query plan in question in more detail, we see that it consists of 6 pipelines; one of them is not part of the longest path, so it is ignored. Two of the pipelines have very small CPU and I/O costs (and also only small numbers of tuples), so they don't affect progress estimation significantly. However, for the remaining 3 pipelines, which make up the majority of the execution, the corresponding weights differ by more than an order of magnitude, with the first two pipelines in the query plan having high weights and each corresponding to nearly half of the execution time in the query (which can be seen by the two distinct 'angles' in the progress estimate in Figure 12); the effect of modelling these weights better (in combination with an initial over-estimate of the cardinality of the third pipeline, whose importance is decreased due to the adjusted weights) results in the improved accuracy of the progress estimates. We will evaluate the overall effects of using operator weights in Section 5.

4.7 Progress Estimation for Data with Column-store Indexes

In addition to the traditional row-at-a-time execution model, Microsoft SQL Server supports processing data in batches of rows when operating on data stored in columnar format, thereby greatly reducing CPU time and cache misses when processing large data sets [16, 17]. These changes affect progress estimation as well, as progress for these operators cannot be based on $GetNext$ calls any more; instead, we use a more coarse-grained approach in which we base the progress estimates for these operators/pipelines on the fraction of column segments processed at any point in time. The dynamic management view *sys.dm_exec_query_profiles* exposes the counters required for computing the numbers of segments processed by an operator as well, while the total number of segments in the underlying index can be computed from the dynamic management view *sys.column_store_segments*.

5. EXPERIMENTS

In this section, we will evaluate the various techniques described in Section 4 using both real-world and synthetic workloads.

Workloads: In order for this evaluation to cover a wide range of query plans and data distributions, we use 5 different workloads; the set of workloads we selected is skewed towards decision-support scenarios, since progress estimation is generally most important for longer-running queries. The workloads we use are

- A 100 GB instance of the TPC-H benchmark, with the data generated with a skew-parameter of $Z = 1$ [1].
- A 100 GB instance of the TPC-DS decision support benchmark.
- "*REAL-1*": a real-world decision-support and reporting workload over a 9GB Sales database. Most of the queries in this workload involve joins of 5-8 tables as well as nested subqueries. The workload contains 477 distinct queries.
- "*REAL-2*" is a different real-life decision-support workload on 12GB dataset with even more complex queries (with a

typical query involving 12 joins). This workload contains a total of 632 queries.

- "*REAL-3*": a 3rd real-life decision-support workload using a 97 GB dataset; this workload contains a total of 40 join and group by queries.

Error Metrics: In the following experiments, we will consider two different error metrics, depending on which aspect of our algorithm we seek to evaluate.

First, in scenarios where we evaluate the accuracy of techniques designed to improve the estimation of the N_i denominators in equation 2, we compare the accuracy of an *unweighted* (i.e., $\forall i : w_i = 1$) progress estimator $Prog$ to the progress estimate we obtain when we replace all estimated N_i with the exact values $\bar{N}_i$ (which we can determine after a query's completion). We measure this error at a number of evenly spaced time-intervals, recording the progress estimates as well as the k_i and N_i for every second of a query's runtime. We use the notation set $Observations(\mathcal{Q})$ to denote the set of all such measurements for a query $\mathcal{Q}$; for simplicity of notation, we denote each measurement using an index $t, t \in Observations(\mathcal{Q})$. For each measurement t, we denote the corresponding number of *GetNext* calls at a node i by k_i^t. We average the resulting error per query over all measurements and over all queries. This measure of accuracy thus becomes:

$$Error_{count} := \frac{1}{|\mathcal{Q}|} \sum_{\mathcal{Q} \in \text{Workload}} \left(\sum_{t \in Observations(\mathcal{Q})} \left| \left(Prog(Q,t) - \frac{\sum_{i \in Nodes(\mathcal{Q})} k_i^t}{\sum_{i \in Nodes(\mathcal{Q})} \bar{N}_i} \right) \right| \right)$$

Second, in practice many users of progress estimates mainly care about how well the estimates correlate with the time required to execute this query. In addition, a number of the techniques discussed in Section 4, such as the selection of different weights w_i for each operator i are aimed at improving the correlation between execution time and progress (and would remain important even if all N_i were estimated without error). Here, for each measurement t we use $Time(t)$ to denote the time the measurement was taken and $Time(t_{start}(\mathcal{Q}))$ to denote the start and $Time(t_{end}(\mathcal{Q}))$ to denote the end of execution for a query $\mathcal{Q}$. We introduce a second error measure that evaluates how well a progress estimator $Prog$ correlates with time, defined as

$$Error_{time} := \frac{1}{|\mathcal{Q}|} \sum_{\mathcal{Q} \in \text{Workload}} \left(\sum_{t \in Observations(\mathcal{Q})} \left| \left(Prog(Q,t) - \frac{Time(t) - Time(t_{start}(\mathcal{Q}))}{Time(t_{end}(\mathcal{Q})) - Time(t_{start}(\mathcal{Q}))} \right) \right| \right)$$

We also consider use operator-specific variants of each error measure, where we only consider the error over all operators of a specific type.

For both metrics, all queries are executed in isolation. Note that for both metrics, the maximum error is small; for $Error_{count}$, it cannot exceed 1.0 and for $Error_{time}$ it cannot exceed 0.5. Hence, even an improvement of 0.1 or even 0.05 is significant in terms of real-life progress estimation accuracy. To illustrate this, we have plotted the progress estimators for two estimators on TPC-DS query 36 in Figure 13, with the difference in error between them being 0.1.

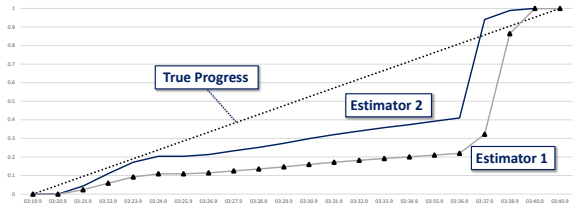

Figure 13: Two example progress estimates with 0.1 difference in Error

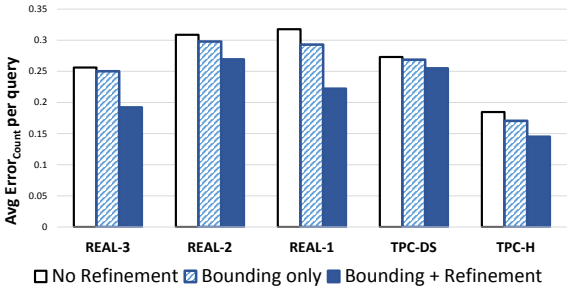

Figure 14: Evaluating the effect of cardinality refinement

5.1 Evaluation of Cardinality Refinement

First, we evaluate the effect of the various techniques used to obtain more accurate estimates of the N_i terms in Equation 2, namely the online cardinality refinement discussed in Section 4.1 and the cardinality bounding described in Section 4.2.

In Figure 14 we show the overall $Error_{count}$ when comparing the overall query progress computed with *exact* knowledge of the correct N_i terms to progress estimates based on three different models: (a) the "Total GetNext" (TGN) Model (which corresponds to Equation 2 with all w_i terms set to 1), (b) the TGN model with cardinality bounding and (c) the query progress model based on *driver nodes* only (DNE), using both online cardinality refinement as well as cardinality bounding. The results are shown in Figure 14. As we can see, both online cardinality refinement as well as bounding both substantially improve the quality of progress estimation across all workloads.

In Figure 15 we show the effect of cardinality refinement broken down by each operator type; we plot the average $Error_{count}$ for each operator when using (a) no refinement, (b) only the refinement based on the techniques described in Section 4.1 and (c) when using the basic cardinality refinement techniques combined with the modifications geared towards addressing the challenges posed by semi-blocking operators (see Section 4.4).

As we can see, while cardinality refinements improve estimates for some operators (most notably *Nested Loop Joins* and the *Bitmap* filter operators associated with *Hash Joins*), they also cause the average accuracy for some operators to degrade, e.g., when the fraction of tuples output by an operator is significantly correlated with time. Still, as shown by Figure 14, even the simple refinement techniques noticeably improve progress estimates when compared to no refinement when averaged over all operators.

Now, adding the techniques that address semi-blocking operators improves the effect of refinement across the board (with only one exception: *Merge Join* operators), in many cases significantly; this illustrates the importance of dealing with this type of buffering, which – in the context of cardinality refinement – has not been addressed in prior work.

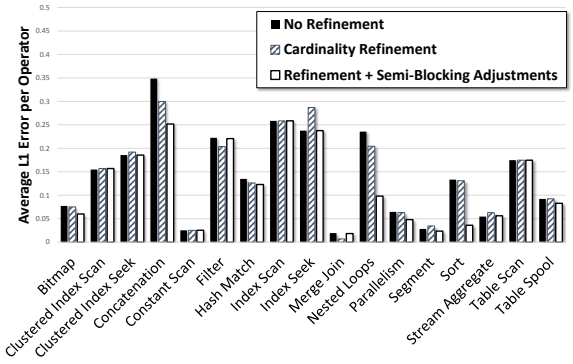

Figure 15: Effect of Cardinality Refinement by Operator

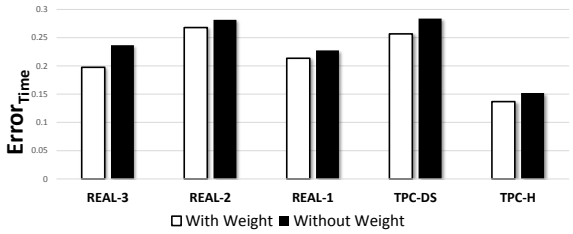

Figure 16: Evaluating the effect of operator weights

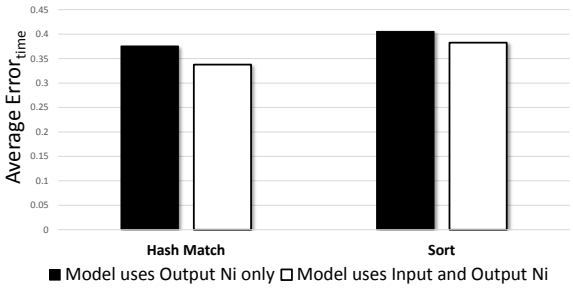

Figure 17: Evaluating the new model for blocking operators

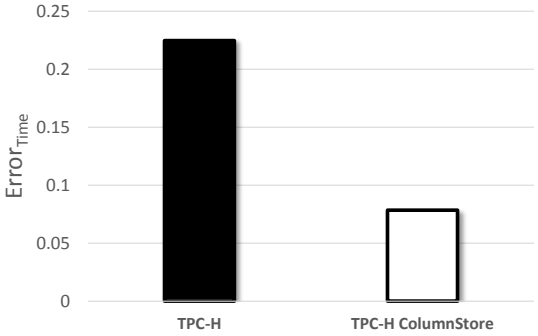

Figure 18: Average Error with and without Columnstore Indexes

5.2 Evaluation of Operator weights

In this experiment, we evaluate the effect of the techniques using the relative operator cost when combining the progress estimates across pipelines on overall per-query accuracy. For this purpose, we evaluated all 5 workloads both with no weights (i.e., $\forall i : w_i = 1$) and compared the resulting accuracy to the weighting scheme described in Section 4.6. Because weights are intended to model the variations in speed across pipelines, we use the $Error_{time}$ metric here. The results are shown in Figure 16. As we can see, applying the weighting scheme improves the correlation between progress estimates and time noticeably across all workloads.

5.3 Evaluation: Improvements for Blocking Operators

In this experiment, we evaluate the effectiveness of the techniques described in Section 4.5 to improve the modelling of blocking operators. For this purpose, we measured $Error_{time}$ for both the original model as well as the improved one over all instances of *Sort* and *Hash Aggregation* operators over all 5 workloads; the results are shown in Figure 17. As we can see, the new model does noticeably improve the modelling of both *Hash* and *Sort* operators; at the same time, the absolute errors are still significant, suggesting that a more refined model of these operators utilizing more information on the internal operator state as future work.

5.4 Evaluation: Progress Estimation for Batch-Mode Operations

In this experiment, we evaluate the effect that the presence of Columnar Indexes has on progress estimation: for this purpose, we executed the TPC-H workload over two different physical designs: (a) the design obtained when building all indexes recommended for this workload by the Microsoft SQL Server Database Tuning Advisor [3] (DTA) and (b) the design obtained where we only construct a non-clustered Columnstore Index on each table in the TPC-H benchmark. The average $Error_{Time}$ over the entire TPC-H workload is shown in Figure 18.

As we can see, the average error is reduced significantly; this turns out to be due to two main factors: for one, the distribution of operators for this workload has changed significantly: in Figure 19 we plotted the counts of each operator across the TPC-H workload for the two different physical designs; as we can see, the non-Columnstore design induces query plans with much more significant variation in the types of operators, whereas the Columnstore design results in query plans which mostly consist of *Index Scans* and *Hash Join* operators and much fewer *Nested Loops* or *Index Seek* operators.

Furthermore, the distribution of progress estimation errors changed significantly as well: in Figure 20, we plot the $Error_{Time}$ broken

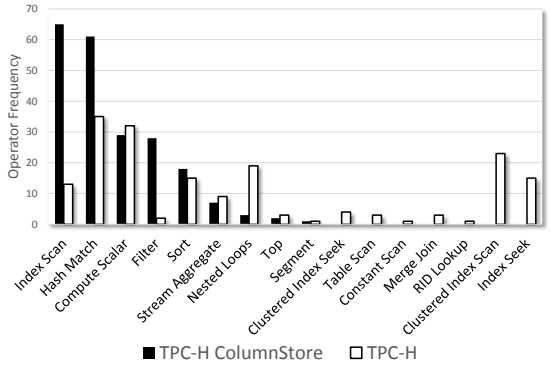

Figure 19: Operator Distribution with and without Columnar Indexes

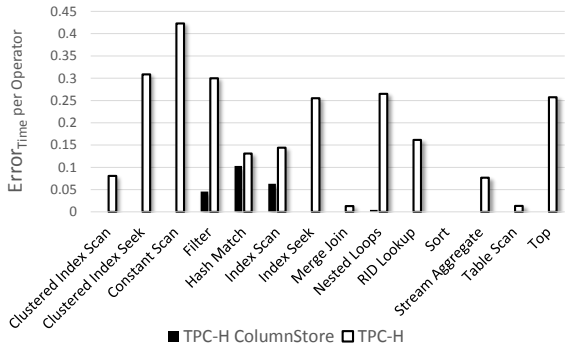

Figure 20: Average Error per Operator with and without Columnar Indexes

down by each operator. We can see that the per-operator error is significantly reduced for all operators occurring in the design based on Columnstore Indexes.

6. RELATED WORK

The problem of progress estimation for database queries was first proposed and studied in [7] and [22], with further refinements in [6, 23, 24]. In addition to the work on query progress estimation itself, there has been considerable research interest in techniques estimating the execution time of queries outright; however, these techniques either require the pre-computation of uniform random samples for all tables touched by a query [29,30], which can be prohibitively expensive in practice, or rely on statistical models trained on prior instances of the same query template and therefore do not generalize to new ad-hoc queries [9, 10].

A statistical approach to progress estimation, that attempts to pick among a set of existing estimators dynamically using machine learning, has been described in [15]. [21] extended query progress estimation to multi-query scenarios, and [19, 25, 26] study progress estimation in *MapReduce* systems, both of which are scenarios that are beyond the scope of this paper.

Other database engines, e.g., *Oracle* expose server side views that provide real time monitoring of actual and estimated cardinality for currently executing query plans similar to the DMVs discussed in this paper. *Oracle* also exposes a dynamic view (*v$session_longops*) that – similar to $dm\_exec\_query\_profiles$ – provides statistical information and progress estimation for database operations such as backup/recovery functions, statistics gathering, and certain types of SQL operators [2]. However, not all operators in SQL queries are covered and therefore (often) no estimate of *overall* query progress is possible.

7. DISCUSSION AND FUTURE WORK

While the experimental evaluation demonstrated the improvements over the earlier state-of-the-art realized as a result of the techniques described in Section 4, they also point to opportunities for further improvements in progress estimates. Because the current design computes a query's progress as part of a client process, most of these improvements require the Microsoft SQL Server engine to expose additional counters beyond the set contained as part of *sys.dm_exec_query_profiles* already. These include:

- More fine-grained information on the internal state of blocking operators such as *Hash* and *Sort*. As illustrated in Figure 11, these operators can perform significant processing that is not concurrent with them outputting rows, meaning that it is not captured well by the *GetNext* model.
- Counters exposing how many tuples are being buffered and how many tuples have been passed on for all semi-blocking operators such as *Nested Loop* joins or *Parallelism*.
- In cases where we have combinations of filters evaluated at a leaf node, we need additional information on the selectivities of these filters: in order to apply the technique described in Section 4.3 in these scenarios, we ideally require the cardinality resulting from the filters that leverage indexes in their execution *only* to adjust the number of logical reads used in the technique described in Section 4.3 accordingly.

Other potential future improvements include (a) the improved propagation of cardinality estimate refinements *across* pipeline boundaries, as we currently only propagate worst-case bounds beyond blocking operators, but not refined cardinalities themselves, and (b) the ability to use feedback from prior executions of queries to adjust the weights that model the relative costs of CPU and I/O overhead when estimating query-level progress, (c) extending the refined model of Section 4.5 to additional operators and (d) improved cardinality-refinement logic for special cases where it is known that the number of *GetNext* calls (per driver node tuple) is highly correlated with time, such as cases of *Merge Joins* where one of the inputs contains large regions of non-joining tuples before the first joining tuple is seen, or filters with predicates on the column the input data is sorted on.

Acknowledgements: We are grateful for the feedback from the anonymous reviewers, which improved this paper significantly.

8. REFERENCES

[1] Program for TPC-H data generation with Skew. ftp://ftp.research.microsoft.com/users/viveknar/TPCDSkew/.

[2] V$SESSION_LONGOPS. http://docs.oracle.com/cd/B19306_01/server.102/b14237/dynviews_2092.htm#REFRN30227.

[3] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB*, pages 1110–1121, 2004.

[4] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A Multidiemsnional Workload-Aware Histogram. In *ACM SIGMOD*, 2001.

[5] S. Chaudhuri. An overview of Query Optimization in Relational Systems. In *ACM PODS*, 1998.

[6] S. Chaudhuri, R. Kaushik, and R. Ramamurthy. When can we trust Progress Estimators for SQL queries? In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 575–586. ACM, 2005.

[7] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Estimating Progress of Execution for SQL queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 803–814. ACM, 2004.

[8] C. M. Chen and N. Roussoploulos. Adaptive Selectivity Estimation Using Query Feedback. In *Proceedings of the ACM SIGMOD Conference*, pages 161–172, May 1994.

[9] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal. Performance Prediction for Concurrent Database Workloads. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 337–348. ACM, 2011.

[10] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. I. Jordan, and D. Patterson. Predicting Multiple Metrics for Queries: Better Decisions enabled by Machine Learning. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 592–603. IEEE, 2009.

[11] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys (CSUR)*, 25(2):73–169, 1993.

[12] R. H. Güting. Operator-Based Query Progress Estimation. Technical Report 343-2/2008, FernUniversität Hagen, February 2008.

[13] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. *ACM SIGMOD Record*, 26(2):171–182, 1997.

[14] A. König and G. Weikum. Combining Histograms and Parametric Curve Fitting for Feedback-Driven Query Result-size Estimation. In *25th International Conference on Very Large Databases*, 1999.

[15] A. C. König, B. Ding, S. Chaudhuri, and V. Narasayya. A Statistical Approach towards Robust Progress Estimation. *Proceedings of the VLDB Endowment*, 5(4):382–393, 2011.

[16] P.-A. Larson, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, S. Rangarajan, R. Rusanu, and M. Saubhasik. Enhancements to SQL Server Column Stores. In *ACM SIGMOD*, 2013.

[17] P.-A. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. Sql Server Column Store Indexes. In *ACM SIGMOD*, 2011.

[18] J. Li, R. Nehme, and J. Naughton. GSLPI: A Cost-based Query Progress Indicator. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 678–689. IEEE, 2012.

[19] J. Li, R. V. Nehme, and J. F. Naughton. Toward Progress Indicators on Steroids for Big Data Systems. In *CIDR*, 2013.

[20] L. Lim, M. Wang, and J. S. Vitter. SASH: a Self-adaptive Histogram Set for Dynamically Changing Workloads. In *VLDB*, 2003.

[21] G. Luo, J. Naughton, and P. Yu. Multi-query SQL Progress Indicators. In *EDBT*, pages 921–941, 2006.

[22] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. Toward a progress indicator for database queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 791–802. ACM, 2004.

[23] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. Increasing the Accuracy and Coverage of SQL Progress Indicators. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 853–864. IEEE, 2005.

[24] C. Mishra and N. Koudas. A Lightweight Online Framework for Query Progress Indicators. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1292–1296. IEEE, 2007.

[25] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: a Progress Indicator for Mapreduce DAGs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 507–518. ACM, 2010.

[26] K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the Progress of MapReduce Pipelines. In *26th International Conference on Data Engineering (ICDE)*. IEEE, 2010.

[27] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO - DB2's Learning Optimizer. In *Proceedings of the 27th Conference on Very Large Databases, Rome, Italy*, 2001.

[28] N. Thaper, S. Guha, P. Indyk, and N. Koudas. Dynamic Multidimensional Histograms. In *Proceedings of ACM SIGMOD Conference, Madison, USA*, pages 428–439, 2002.

[29] W. Wu, Y. Chi, H. Hacígümüş, and J. F. Naughton. Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads. *Proceedings of the VLDB Endowment*, 6(10):925–936, 2013.

[30] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigumus, and J. F. Naughton. Predicting Query Execution Time: Are Optimizer Cost Models really Unusable? In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 1081–1092. IEEE, 2013.

APPENDIX

A. CARDINALITY BOUNDING LOGIC

In this section, we will describe the logic used to the compute worst-case bounds on cardinalities during query execution described in Section 4.2. The following table describes how we compute the lower bound LB_i and upper bound UB_i for a node i, broken down by the type of logical operator node i corresponds to. We use the offset $i-1$ to denote the direct child of node i. For *Nested Loops* and *Hash joins*, we use $i-1$ to denote the outer and $i-2$ the inner child. Finally, for nodes with variable number of children, we use the offset j to iterate over all of them (i.e., $\sum_j$).

Logical Operator	$\mathbf{LB_i}$	$\mathbf{UB_i}$
Inner Join	K_i	$(UB_{i-1}K_{i-1}) \cdot UB_{i-2} + K_i$
Left Anti Semi Join	K_i	$(UB_{i-1}K_{i-1}) \cdot UB_{i-2} + K_i$
Left Semi Join	K_i	$(UB_{i-1}K_{i-1}) \cdot UB_{i-2} + K_i$
Right Outer Join	K_i	$(UB_{i-1}K_{i-1}) \cdot UB_{i-2} + K_i$
Right Semi Join	K_i	$(UB_{i-1}K_{i-1}) \cdot UB_{i-2} + K_i$
Full Outer Join	K_i	$(UB_{i-1}K_{i-1}) \cdot UB_{i-2} + K_i$
Concatenation	$\sum_j K_j$	$\sum_j UB_j$
Clustered Index Seek Index Seek Index Scan	K_i	$TableSize$, or, when on inner side of join: $TableSize \cdot UB_{i-1}$
Table Scan	Table Size	$TableSize$
Constant Scan	N_i	N_i
Eager Spool Lazy Spool	K_i	∞
Filter	K_i	$(UB_{i-1}K_{i-1}) + K_i$
Distribute Streams	K_i	$(UB_{i-1}K_{i-1}) + K_i$
Gather Streams	K_i	$(UB_{i-1}K_{i-1}) + K_i$
Repartition Streams	K_i	$(UB_{i-1}K_{i-1}) + K_i$
Segment	K_i	$(UB_{i-1}K_{i-1}) + K_i$
Distinct Sort	K_i	$(UB_{i-1}K_{i-1}) + K_i$
Sort	K_{i-1}	UB_{i-1}
Top N Sort	K_{i-1}	$\min\{N, UB_{i-1}\}$
Bitmap Create	K_{i-1}	UB_{i-1}
Aggregate Partial Aggregate	$\max(1, K_i)$	$UB_{i-1} - \max(1, K_i)$
Compute Scalar	K_{i-1}	UB_{i-1}
Top N Sort	K_{i-1}	$\min\{N, UB_{i-1}\}$
RID Lookup	K_i	UB_{i-1}
Eager Spool Lazy Spool	LB_{i-1}	UB_{i-1}, or, when on inner side of join: $UB_{i-1} \cdot UB_{i-2}$

Table 1: Bounding logic for each operator