INFORMATIK
BERICHTE

343 - 2/2008

Operator-Based Query Progress Estimation

Ralf Hartmut Güting

FernUniversität in Hagen

Fakultät für Mathematik und Informatik
Postfach 940
D-58084 Hagen

Operator-Based Query Progress Estimation

Ralf Hartmut Güting
LG Datenbanksysteme für neue Anwendungen
Fakultät für Mathematik und Informatik, Fernuniversität Hagen
D-58084 Hagen, Germany, rhg@fernuni-hagen.de

Abstract: Recently, research has addressed the problem of estimating progress for long-running database queries. The basic idea is to "continuously" monitor execution to keep track of how much work has been done, and at the same time to collect statistics to arrive at a more and more refined estimate of the total amount of work that is needed. Previous research has generally decomposed the operator tree for the query into pipelines (or "segments") of non-blocking operators, tried to observe progress per pipeline and then to combine progress measures of the different pipelines into an overall progress measure. It has soon become apparent that pipelines of non-blocking operators are too large units and that it is necessary to define smaller segments (e.g. containing only one join operator).

In this paper we take a more radical approach where each operator in a query tree is able to estimate the progress achieved for its subtree based on the progress reported by its children. No global analysis of the query tree is needed, nor is it necessary to determine driver nodes or dominant inputs. Each operator is strictly independent in its progress estimation. Nevertheless progress estimation works fine across blocking operators and for the whole query tree. The technique lends itself to a simple and clean implementation. It is suitable for extensible database architectures where the set of query processing operators is large and possibly extended at any time. Our implementation allows one to add progress support for operators gradually such that the system runs at any time and reports progress whenever all operators in the query tree support progress. We report a prototypical implementation in the SECONDO extensible database system. Progress estimation now is a standard feature of SECONDO. To our knowledge it is the first freely available DBMS prototype that includes query progress estimation.

1 Introduction

For many kinds of long-running software tasks, human computer user interfaces include a visualization of the progress made, usually in the form of a progress bar with an estimate of the remaining time. Some examples of such tasks are file download, burning a CD-ROM, or installing an operating system. For a long-running database query such a progress indication would also be quite useful. The user can plan her time better, e.g. go for lunch instead of waiting for the completion of the query. If unexpectedly slow progress is shown, this can be a hint that the query was wrongly formulated. Based on this one may cancel the query execution. For a long running task it is also comforting to see that progress is being made, albeit slowly, and that the system is not stuck.

Perhaps surprisingly, progress indicators for database queries are not yet in common use. The most likely reason is that the problem is difficult. Progress is the fraction of work performed relative to the total amount of work to be done. Whereas it is easy to observe (or count) the work that has been done, the estimation of the total amount of work is a difficult problem as is well known from query optimizers. Simple solutions, such as estimating the remaining time as the difference between the optimization total time estimate and the elapsed time, do not work, as the optimization time estimate is not precise enough and can be quite wrong.

Recent research [2, 3, 6, 7, 8] has addressed the problem of progress estimation for long-running queries. The general idea is to observe execution at regular time intervals, counting the work that has been done (e.g. the number of tuples processed by operators) and collecting statistics that allow one to continuously improve the estimate for the total (or remaining) work needed. A main problem in query optimization is the prediction of the cardinality of intermediate results, produced by operators implementing selection or join, but also others such as groupby, for example. During execution one may observe the selectivity of a selection or join as the fraction of tuples returned relative to those processed. Hence cardinality estimation and the estimate of total work can be continuously improved.

Characteristic for all the research mentioned above are the following apects: First, an operator tree representing a query plan is decomposed into pipelines of non-blocking operators. Pipelines are interesting because they have one of three states, namely (a) completed, (b) currently executing, and (c) not yet running, and all operators in a pipeline share this state. Progress is determined per pipeline and then summed up to determine the overall progress. Second, operators in a pipeline work synchronously so that one can observe progress of a pipeline possibly at a single point in the pipeline. So these works try to identify so-called “driver nodes” or “dominant inputs” as observation points. Third, the cost measure for processing in a pipeline is kept simple. One approach uses the total numbers of tuples returned by operators, the other pages of bytes read or output at pipeline boundaries.

Whereas these approaches achieve satisfactory results in many cases, there are also some problems:

1. It turns out that in some cases, pipelines are too coarse units. So [7] observe that in order to make accurate predictions it is necessary to split pipelines (called segments there) into smaller parts ensuring that a segment contains only one join operator (or other set operator). However this blurs the initial idea of pipelines and makes concepts more and more complicated.
2. The simple cost measure makes it impossible to balance costs adequately between different pipelines, especially currently executing and not yet started ones.

To illustrate this, suppose there are two pipelines P_1 and P_2 separated by a blocking operator. Assume that the per tuple cost in P_2 is 1 and that in this case both pipelines require the same amount of time T for processing. Hence when P_1 completes, overall progress is 50%. Let us assume the progress estimator reports this correctly.

However, now assume the per tuple cost in P_2 is 10, for example, because an expensive predicate is evaluated for each tuple. Hence the total time for P_2 is $10T$. It does not help that in P_1 the number of tuples to be processed in P_2 is estimated precisely. As in the first case, assuming a uniform tuple cost of 1, progress at completion of the first pipeline will be estimated as 50% whereas the real progress is less than 10%.

In this paper, we present a different approach that addresses the problems above. Our approach was motivated by the goal of supporting progress estimation in an extensible database prototype, SECONDO. The SECONDO system is a nice environment for studying and implementing progress estimation techniques as it is an open source system that is freely available for download [11] and central components like the optimizer and the query processor are accessible and relatively well documented. Further documentation about SECONDO can be found at [11].

SECONDO has a modular architecture and supports complex non-standard data types and operations (e.g. for moving objects). This leads to the following special requirements.

- Query processing is organized as a set of algebra modules each providing certain operations. It should be possible to add new query processing operators without any need to update global modules for progress estimation.
- The system allows one to enter query plans directly and a lot of experimentation is done in this mode, i.e., without using the optimizer. Hence progress estimation should work as well as possible in this case (even though cost estimation for not yet running operators cannot be precise without help of the optimizer).
- Progress estimation must support expensive predicates.

As a result, our approach has the following novel features:

- Progress estimation is based on the idea that each operator estimates the total time required for processing its subtree (the one of which it is the root) and the progress achieved for this subtree, based on similar estimates by its children. We will show that this works across blocking operators, so no definition of pipelines or global analysis of the query tree is needed. Each operator is strictly independent in its progress estimation and can perform it locally. Progress of the query is the progress reported by the root operator.
- Cost estimation for each operator can be done as precisely as desired. For example, per tuple and per byte costs can be distinguished. In particular, optimizer information about expensive predicates can be utilized. This means that cost can be balanced correctly between active and passive operators.
- Progress estimation works, although not perfectly, for query plans that do not come from the optimizer.
- For the first time, to our knowledge, progress estimation is studied for operators that cut off a stream of tuples from the end (e.g. applying a *first n* clause after a big SQL query). This is achieved by explicitly modeling *blocking time* and *blocking progress* for operators.
- We provide an analysis of the reliability of on-line selectivity estimates of filter and join operators.
- The overhead is extremely low.
- To our knowledge SECONDO is the first freely available DBMS prototype that includes query progress estimation.

The paper is structured as follows. Section 2 explains the concepts and illustrates operator based progress estimation, providing precise progress formulas for a small set of basic operators. The implementation in SECONDO is outlined in Section 3. Strategies for progress estimation for a larger set of operators are discussed in Section 4. Experimental results are shown in Section 5. Several important open questions are discussed in Section 6. Related work is described in Section 7. Finally, Section 8 concludes the paper.

2 Operator-Based Progress Estimation

2.1 Overview

Answering Progress Queries. The basic idea of operator based progress estimation is that each operator in an operator tree representing a query plan is able to estimate the total time needed for processing its subtree (e.g. in milliseconds) and the relative progress already achieved for this subtree (a number between 0 and 1). To determine these numbers, an operator can ask each of its predecessors (children in the operator tree producing its argument streams) for the following quantities[1] expected for their respective subtree:

- *cardinality*: the total number of tuples
- *size*: the average tuple size
- *time*: the total time for processing the subtree
- *progress*: the relative progress achieved in processing the subtree

Furthermore, the operator maintains counters to keep track of the relevant numbers of steps in its own implementation to determine its own relative progress; in most cases this will be the numbers of tuples read from the input streams and returned on the output stream. Based on this information, the operator then on request returns the same quantities to its successor (parent in the operator tree).

Asking a predecessor for progress information we call sending a *progress query*, and returning such information answering a progress query.

Cold and Warm States. Estimation of result cardinalities and tuples sizes for some operators depend on observed tuple flow through this operator. For example, a *filter* operator evaluates a predicate on each tuple in a stream and observes its selectivity as the fraction of tuples returned. The same holds for join operators that observe the number of tuples returned relative to the number of pairs considered.

A similar problem occurs for operators computing new attributes whose values can be of varying size. For example, think of an operator that computes polygons as intersections of two polygons. The average size of the new attribute and hence, the average tuple size, can only be observed when tuples flow through the operator.

However, when there are blocking operators in a query plan (e.g. *sort*), then for considerable periods of time no tuples flow through the operators higher up in the operator tree. We say, an operator is in *cold state*, if no tuples are flowing through it yet (or too few to make a good prediction), and in *warm state*, when it has processed a sufficient number of tuples.

Such operators will then use different estimation strategies for the warm and the cold state. In the warm state, estimation is based on observations. In the cold state, prediction is based on either defaults, or, if available, on selectivity information passed from the optimizer. To this end, we make such information available in query plans.

Interaction With the Query Processor. The query processor at regular time intervals, say, every 0.1 seconds, sends a progress query to the root of the operator tree. This operator will recursively send progress queries to its predecessor(s) and on receipt of the answer(s) produce progress information and return it to the query processor. The expected time and the relative progress can then be passed to a user interface for display.

Communication between operators is done via the query processor. An operator implementation can call a method *RequestProgress* of the query processor to get the progress information of its first argument, for example. The query processor then calls a method *RequestProgress* of the respective operator. An operator is provided an address of a *ProgressInfo* record by the query processor which has fields for the mentioned quantities; by setting these fields it can return the values.

From the point of view of the query processor, evaluation of progress queries is quite similar to the regular evaluation of queries; just for each operator instead of the standard evaluation method a *Request-*

1. The set of quantities will be refined later.

Progress method is called. In SECONDO, this change is particularly easy, since even the standard evaluation procedure can be used, just passing a different message to operator implementations.

Decision to Support Progress for Each Operator. We allow for each operator an independent decision to support progress or not. In this way, in a large existing system that already has lots of query plan operators, progress estimation can be added gradually. The system will work at all times. To this end, an operator can individually register to support progress queries. The query processor, when asked by an operator to evaluate progress for one of its arguments, checks whether that operator supports progress. It returns *false* if the operator does not support progress and *true* if it does.

Accordingly, an operator implementing progress estimation must always expect that an argument does not yield progress information. In this case, most likely it cannot produce a progress estimation itself. It then returns a *Cancel* message to the query processor. It returns *Yield*, if it is able to compute progress information. The query processor converts such messages into a boolean that it returns to the caller.

The query processor will trigger display of progress at the user interface when it receives the first progress information from the root operator. Hence, if an operator tree contains operators that do not support progress, the progress display will simply not appear.

Decision to Answer for Each Progress Query. Since progress queries are posed by the query processor in an asynchronous manner, a working operator must expect progress queries at any arbitrary time. Usually operators work in stream mode, that is, they are called according to a protocol[2]

```
Open Request* Close
```

Operators supporting progress must now be prepared to handle progress queries at any time, even before the *Open* or after the *Close* message. This may be a problem, since on *Open* the data structures are initialized that are needed to answer progress queries. Therefore, operators are always allowed to return *Cancel* when they are not yet capable to answer progress queries. This does not mean that progress estimation does not work for this query; it simply means that at this instant of time it is not available. The query processor will ask again later.

2.2 Progress Estimation for Some Basic Operators

In this subsection, we illustrate progress estimation for a few basic operators. A larger set of operators is discussed in Section 4. The operators considered here are the following:

```
feed:       rel(Tuple) -> stream(Tuple)
filter:     stream(Tuple) x (Tuple -> bool) -> stream(Tuple)
consume:    stream(Tuple) -> rel(Tuple)
count:      stream(Tuple) -> int
product:    stream(Tuple1) x stream(Tuple2) -> stream(Tuple3)
symmjoin:   stream(Tuple1) x stream(Tuple2)
              x (Tuple1 x Tuple2 -> bool) -> stream(Tuple3)
```

In the next subsection we discuss

```
head:       stream(Tuple) x int -> stream(Tuple)
```

The meaning of most operators should be well-known; it will also be explained as we go through them. The progress information passed between operators is now refined. We pass a structure containing the following information:

2. Request is called GetNext() in standard terminology.

- *C* - Cardinality: Expected total number of tuples returned from this subtree
- *S* - Size: Expected average tuple size [bytes]
- *SN* - Size: Number of attributes
- *SV* - Size: For each attribute, the expected average size [bytes]
- *T* - Time. Expected total time for processing this subtree [ms]
- *P* - Progress. Fraction of expected time for work already done in this subtree relative to expected total time [0, 1]

In addition to what was mentioned before we also need the number of attributes and the average size per attribute. They are necessary to determine the new tuple size after a projection.

For the counters inserted into the code of an operator we use the following notations:

- *m* - the number of tuples returned (passed to the successor)
- *k* - the number of tuples read (if there is only one argument stream)
- k_1, k_2 - the numbers of tuples read from the first and second argument stream, respectively.

We refer to progress information passed by one of the argument operators by indices, e.g. C_1 and C_2 are the cardinalities returned by progress estimation of the first and second argument, respectively. The size related quantities together (*S*, *SN*, *SV*) we denote as *Sizes, e.g.* $Sizes_1$ and all quantities together as *Progress*, e.g. $Progress_2$. This is useful, as for some operators one can copy the size information or even all fields. Similarly, given size information from two arguments, a function $JoinSizes(Sizes_1, Sizes_2)$ computes result sizes for any kind of join operator in the obvious way.

An operator implementation can get a selectivity value and a predicate cost value from its node in the operator tree, via a request to the query processor denoted as *qp.Sel* and *qp.PredCost* respectively. These values are set to 0.1 (for selectivity) and 0.1 milliseconds (for predicate cost) by default (see Section 6.2 for a discussion of these constants). If the query plan was constructed by the optimizer, it was annotated with the values determined in optimization, and this annotation has been entered into the operator tree (see Section 3).

Operator *feed*. The *feed* operator scans a relation and produces a tuple stream. Usually the relation is stored on disk. In this case, progress information is computed as shown in Figure 1. Cardinalities and

$$\boxed{\begin{aligned}
C_{feed} &= R.Card + 1 \\
S_{feed} &= R.Size \\
SN_{feed} &= R.NoAttrs \\
SV_{feed} &= R.Size[i], \text{ for } i = 1, \ldots, SN_{feed} \\
T_{feed} &= C_{feed} \cdot (u_{feed} + S_{feed} \cdot v_{feed}) \\
P_{feed} &= m \cdot (u_{feed} + S_{feed} \cdot v_{feed}) \,/\, T_{feed} = m \,/\, C_{feed}
\end{aligned}}$$

Figure 1: Progress estimation for *feed*, stored relation

tuple and attribute sizes are read from the relation *R*.[3] We add one to the cardinality to make sure it is not zero.[4] The total time required by the *feed* operator is based on the cardinality and the tuple size (as tuples are read from disk), using constants u_{feed} and v_{feed} to denote time overhead per tuple and time per byte. These constants are determined in experiments. The progress is the time estimated for processing the *m* tuples that have been returned already divided by the total time, which in this case simplifies to the obvious progress measure $m \,/\, C_{feed}$.

3. Such information may be obtained from the system catalog; in SECONDO one can ask the relation object for it.
4. Zero cardinalities must be avoided as they may lead to expected time $T = 0$. Computation of progress *P* divides by *T*. Adding 1 is a simple solution that introduces only negligible errors.

However, it is also possible that the argument relation is built on the fly, e.g. by a *consume* operator.[5] This case is shown in Figure 2. Here cardinalities and sizes are received from the *consume* operator dur-

$$
\boxed{\begin{aligned}
C_{feed} &= C_1 \\
S_{feed} &= S_1 \\
SN_{feed} &= SN_1 \\
SV_{feed} &= SV_1 \\
T_{feed} &= T_1 + C_{feed} \cdot (u_{feed} + S_{feed} \cdot v_{feed}) \\
P_{feed} &= (P_1 T_1 + m \cdot (u_{feed} + S_{feed} \cdot v_{feed})) \,/\, T_{feed}
\end{aligned}}
$$

Figure 2: Progress est. for *feed,* relation built by *consume*

ing or after building the relation. The total time required is the sum of that of the *consume* subtree and the time needed by *feed* itself. Observe that progress grows continuously through the time of completing the construction of the relation; before this time we have $P_1 < 1$, $m = 0$ and afterwards $P_1 = 1$, $m > 0$. Adding evaluation times for operators and observing progress within each part allows one to nicely balance between different operator costs.

Operator *filter*. The *filter* operator evaluates a predicate on each tuple of a stream and returns the tuples fulfilling the predicate. Progress estimation is shown in Figure 3 for the cold state.

$$
\boxed{\begin{aligned}
C_{filter} &= C_1 \cdot qp.Sel \\
Sizes_{filter} &= Sizes_1 \\
T_{filter} &= T_1 + C_1 \cdot qp.PredCost \cdot u_{filter} \\
P_{filter} &= (P_1 T_1 + k \cdot qp.PredCost \cdot u_{filter}) \,/\, T_{filter}
\end{aligned}}
$$

Figure 3: Progress estimation for *filter*. Cold state: $m < 50$

As long as the operator has not processed a sufficient number of tuples, its own selectivity estimation may not be reliable, and we prefer to use an estimate from the optimizer instead (if available), or a default value. We assume that a stable estimation is reached when the operator has returned at least 50 tuples and then switch to the operator's observed selectivity (Figure 4). See Section 6.2 for a discussion of constant 50 as a threshold for the warm state.

$$
\boxed{C_{filter} = C_1 \cdot m \,/\, k}
$$

Figure 4: Progress est. for *filter*. Warm state: $m \geq 50$

Operator *consume*. The *consume* operator collects a stream of tuples into a relation. Progress estimation is shown in Figure 5.

$$
\boxed{\begin{aligned}
C_{consume} &= C_1 \\
Sizes_{consume} &= Sizes_1 \\
T_{consume} &= T_1 + C_1 \cdot (u_{consume} + S_1 \cdot v_{consume}) \\
P_{consume} &= (P_1 T_1 + k \cdot (u_{consume} + S_1 \cdot v_{consume})) \,/\, T_{consume}
\end{aligned}}
$$

Figure 5: Progress estimation for *consume*

This is straightforward; *u* and *v* are again per tuple and per byte constants.

Operator *count*. The *count* operator counts the number of tuples in a stream. Progress estimation is shown in Figure 6.

5. A ... *consume feed* ... sequence normally does not make a lot of sense, but it is allowed. In the context of progress estimation it is interesting to study it as the most simple case of a blocking operator in a query plan.

$$\boxed{Progress_{count} = Progress_1}$$

Figure 6: Progress estimation for *count*

The processing time for *count* itself is negligible compared to other operators in a query plan, and the only one asking *count* for progress information will be the query processor. Hence we simply copy all progress information from the predecessor.

Operator *product*. The *product* operator computes the Cartesian product of two streams of tuples. It first reads the second stream entirely into a buffer. It then starts processing the first stream, combining each tuple with the complete contents of the buffer. Progress estimation is shown in Figure 7.

$$\boxed{\begin{array}{lll} C_{product} & = & C_1 \cdot C_2 \\ Sizes_{product} & = & JoinSizes(Sizes_1, Sizes_2) \\ T_{product} & = & T_1 + T_2 + C_2 \cdot u_{product} + C_1 \cdot C_2 \cdot v_{product} \\ P_{product} & = & (P_1T_1 + P_2T_2 + k_2 \cdot u_{product} + m \cdot v_{product}) \mathbin{/} T_{product} \end{array}}$$

Figure 7: Progress estimation for *product*

Cost estimation for this operator shown in this section is a bit simplistic as it does not take tuple sizes into account. This is valid only if the second stream fits entirely into the memory buffer; costs per byte would arise if the buffer overflows on disk. Of course, one can make these cost formulas more sophisticated by modeling in which cases disk accesses are needed.

Operator *symmjoin*. Finally, we discuss *symmjoin* as a symmetric, non-blocking, nested loop join operator. It maintains two buffers *A* and *B*, one for each input stream. In each step, it reads a tuple *a* from the first input stream into buffer *A* and evaluates the join predicate against all tuples in buffer *B* returning successful matches; then it reads a tuple *b* into buffer *B* and matches against all tuples in *A*. When one stream is exhausted, it reads the remaining tuples from the other stream (without writing to a buffer), matching against the buffer of the exhausted stream. Progress estimation is shown in Figure 8 and Figure 9.

$$\boxed{\begin{array}{lll} C_{symmjoin} & = & C_1 \cdot C_2 \cdot qp.Sel \\ Sizes_{symmjoin} & = & JoinSizes(Sizes_1, Sizes_2) \\ T_{symmjoin} & = & T_1 + T_2 + C_1 \cdot C_2 \cdot qp.PredCost \cdot u_{symmjoin} \\ P_{symmjoin} & = & (P_1T_1 + P_2T_2 + k_1 \cdot k_2 \cdot qp.PredCost \cdot u_{symmjoin}) \mathbin{/} T_{symmjoin} \end{array}}$$

Figure 8: Progress est. for *symmjoin.* Cold state: $m < 50$

Similarly as for the *filter* operator, we rely on the operator's selectivity estimate only after a sufficient number of tuples have been returned.

$$\boxed{C_{symmjoin} = C_1 \cdot C_2 \cdot m \mathbin{/} (k_1 \cdot k_2)}$$

Figure 9: Progress est. for *symmjoin.* Warm state: $m \geq 50$

As for *product*, tuple sizes are not taken into account which only holds if memory buffers do not overflow.[6]

As a general remark, note that somewhat wrong cost estimates are not as disastrous as in query optimization. In progress estimation, they only lead to a wrong balance of weights between different opera-

6. In the SECONDO implementation of join operators, tuples are not copied when forming a result tuple; only pointers to attributes are copied. Tuple sizes will play a role later if the result stream is written to disk by *consume*, but not if result tuples are filtered away or counted, for example.

tors. This in turn will lead to progress being observed at varying speeds, which is not ideal but often tolerable.

2.3 An Extension: Supporting Cutting Off Streams

The framework described so far supports many query processing operators, in fact, all we can think of, except one: the *head* operator. This operator is unique in that it cuts off a stream from the end. Here is again the signature:

```
head: stream(Tuple) x int -> stream(Tuple)
```

The *head* operator takes a stream of tuples and an integer *n* and only returns the first *n* tuples of the argument stream. The problem is that it obviously needs to know to what extent its argument stream is produced by blocking operators. Consider the following two example queries:

```
(1) R feed head[1000] count
(2) R feed sortby[A asc] head[1000] count
```

Suppose *R* is a relation with a million tuples and a progress query occurs when *head* has received 500 tuples. For query (1), the progress information returned by the *feed* operator will indicate a large total time *T* and a small progress value *P*, namely, 500 / 1000000 = 0.0005. However, *head* should be able to estimate that already 50% of the work is done, and that the total time is very small.

In contrast, for query (2), the progress information returned by the *sort* operator looks quite similar to that delivered by *feed*, namely, a large total time *T* and a very small progress *P* achieved so far. By the fact that it has received 500 tuples, *head* might conclude again that its progress is 50%. This would be quite wrong, as the long blocking time of the sort operator has passed and the remaining 500 tuples will be delivered very quickly. Hence the actual progress is more like 99%.

To solve this problem, we add two more fields to the progress information passed between operators, namely *blocking time* and *blocking progress*.

- *BT* - Blocking Time. The expected time required before this subtree returns the first tuple [ms]
- *BP* - Blocking Progress. Fraction of expected time for work already done within the blocking phase relative to expected total blocking time.

For the operators discussed so far, their progress estimations are extended as follows (Figure 10).

Operator	*Blocking Time*	*Blocking Progress*
feed (stored rel.)	0.001	1.0
feed (rel. built by consume)	BT_1	BP_1
filter	BT_1	BP_1
consume	T_{consume}	P_{consume}
count	BT_1	BP_1
product	$BT_1 + BT_2 + C_2 \cdot u_{product}$	$(BP_1 \cdot BT_1 + BP_2 \cdot BT_2 + k_2 \cdot u_{product}) / BT_{product}$
symmjoin	$BT_1 + BT_2$	$(BP_1 \cdot BT_1 + BP_2 \cdot BT_2) / BT_{symmjoin}$

Figure 10: Blocking time and blocking progress

Feed on a stored relation does not block; hence the blocking time is 0 and the blocking progress 1. As mentioned before, zero time estimates must be avoided; hence we use 0.001 instead. The *feed* operator on a constructed relation, *filter*, *count*, and *symmjoin* are non-blocking; they all just need to pass the blocking time and progress values of their predecessors. *Consume* is totally blocking; hence its time and progress estimates are also estimates for blocking time and blocking progress. *Product* is blocking while reading the second argument stream into the buffer; hence it combines its blocking time and progress with that of the predecessors.

Operator *head*. Since now blocking information is available, we can define progress estimation for the *head* operator as shown in Figure 11.

$$\boxed{\begin{array}{lcl} C_{head} & = & \min(\, n, C_1) \\ Sizes_{head} & = & Sizes_1 \\ T_{head} & = & BT_1 + C_{head} \cdot (perTuple + u_{head}) \\ & & \text{where } perTuple = (T_1 - BT_1) \,/\, C_1 \\ P_{head} & = & (BP_1 BT_1 + m \cdot (perTuple + u_{head})) \,/\, \mathrm{T}_{\mathrm{head}} \\ BT_{head} & = & BT_1 \\ BP_{head} & = & BP_1 \end{array}}$$

Figure 11: Progress estimation for *head*

Here *n* denotes the value of the second argument of *head*. The estimated total time is the blocking time of the argument stream plus the time needed after the blocking phase. The latter time is for each tuple returned the per tuple time of the argument stream (after blocking) plus a constant for *head* itself. Like other non-blocking operators, *head* simply passes blocking time and progress from its predecessor.

3 Some Implementation Issues

The framework described above has been implemented in the SECONDO prototype. An outline of the implementation strategy was given in Section 2.1. In this section we provide some more details on the implementation done in SECONDO.

Extending the Stream Protocol. The standard protocol for stream processing has the form

```
Open Request* Close
```

On *Open*, initializations are done and required data structures allocated. Then, in a loop *Request* messages are sent and stream elements returned (usually tuples) together with a message *Yield* or *Cancel*. On *Close*, terminating actions are performed and any data structures deallocated.

We have introduced two new messages called *RequestProgress* and *CloseProgress*. *RequestProgress* is triggered asynchronously by the query processor (see below) and hence can occur at any time, possibly before the *Open* or after the *Close* message. Since *Open* initializes the data structures that are also needed for answering *RequestProgress* messages, in the first case the operator cannot return progress information and replies *Cancel*. However, it is necessary that progress queries can be answered after the *Close* message. For example, a blocking operator may close its input stream; this does not mean that the query is completed. Hence we cannot deallocate data structures any more in the *Close* message (at least not the parts used for progress queries). Furthermore, observe that streams can be used in a loop (e.g. in the *loopjoin* operator explained in Section 4). Therefore, we have extended the stream protocol as follows:

```
(Open Request* Close)* CloseProgress
```

The *CloseProgress* message is sent only once by the query processor after complete evaluation of the query. Hence on *CloseProgress* any (remaining) data structures can and will be deallocated. Because some data structures are now not deallocated on *Close* but are allocated on *Open*, the implementation of the *Open* method is changed such that first a check is performed whether a data structure exists; if so, it is deallocated. Then a new instance of the data structure is created. In this way any storage leaks are avoided.

Evaluating Progress Queries. Query processing in SECONDO is done in cooperation between an *Eval* method of the query processor and operator implementation functions. Essentially, *Eval* is applied to a node of the operator tree and returns the value of that subtree. If the node is a leaf of the tree, it returns the value directly. If the node represents an operator applied to some arguments, *Eval* recursively evaluates all sons that do not represent stream or function arguments (called *evaluable* arguments) and puts the results into an argument vector for the operator. For the stream and function arguments it just puts the pointers to the subtrees into the argument vector. Then the operator function is called, passing the argument vector and a message from the set *Open*, *Request*, *Close*. For simple operators that do not return streams (e.g. integer addition), the message is ignored.

Operator implementation functions can take their evaluable arguments from the argument vector. For stream and function arguments, they explicitly ask the query processor for evaluation, calling query processor methods *qp.Open*, *qp.Request*, *qp.Close* and *qp.Received*. Operators handling different messages branch into a part of their code for that message.

This scheme is very easily extended by adding the new messages *RequestProgress* and *CloseProgress*. Operators can send these messages to their arguments using new query processor functions *qp.RequestProgress* and *qp.CloseProgress*. Before calling *Eval* with the respective message for a node, these functions check whether this operator was registered to support progress. Operator implementation functions just add additional branches in their code to process the two new message types.

Embedding Progress Queries into Evaluation. To avoid more complicated multi-threading schemes, we have embedded the triggering of progress queries into the *Eval* function of the query processor. In processing tuple streams, the *Eval* function is called at least once for each tuple (possibly several times, e.g. if predicates are evaluated). The inserted pseudocode looks like this:

```
progressCtr--;
if progressCtr == 0 then
  if currentTime() - lastTime > progressTimeInterval then
    if RequestProgress(QueryTree, ProgressInfo) then
      ModifyProgressView(ProgressInfo)
    endif;
    lastTime = currentTime()
  endif;
  progressCtr = startValue
endif
```

In the current implementation, *startValue* = 100 and *progressTimeInterval* = 100 (clock ticks ≈ 0.1 seconds). Hence, on each call of *Eval*, a counter is decremented, and on each 100th call, the system time is checked. So only very little overhead is added to control the triggering of progress queries.

Making Optimizer Information Available in Query Processing. The optimizer routinely determines selectivity and predicate cost for each selection or join predicate. This information should be available to operator functions. To this end, the syntax for query plans was extended by an annotation of the form {*selectivity*, *cost*}. For example, we may write a query plan

```
Cities feed filter[.Population > 100000] {0.00234, 0.088} consume
```

In SECONDO, operators are often applied in postfix notation. Here *feed* is applied to *Cities*, yielding a stream of “city” tuples, *filter* is applied to this stream, and *consume* collects the stream into a relation. The `{0.00234, 0.088}` annotation contains selectivity and cost for the *filter* operator.

The parser was extended to translate this to a pseudo-operator *predinfo*. The SECONDO parser generally translates text syntax to nested list syntax, hence to

```
(consume (predinfo 0.00234 0.088 (filter (feed Cities) (fun ...))))
```

The query processor, after annotating the query in nested list syntax with further information, constructs the operator tree. This was extended to handle the *predinfo* pseudo-operator in such a way that the selectivity and predicate cost values are written into corresponding fields of the node for the third argument (the *filter* node in this example). The query processor was extended by methods for getting and setting these fields to be used by operator functions. Finally, the optimizer was extended to include this annotation for each operator implementing selection or join when constructing the query plan.

4 Progress Estimation for Different Classes of Operators

In this section we discuss progress estimation strategies and implementation techniques for a larger set of operators, but without giving precise formulas (to keep the space limited). For all signatures shown, progress estimation has been implemented in the SECONDO prototype. Note that the purpose of the section is to illustrate how progress estimation can be realized for various kinds of operators, not necessarily to justify each single aspect of SECONDO query processing or the choice of operators available.

Simple Operators. Some operators either simply let a stream of tuples pass through without changing tuple size or number, or are a sink for a tuple stream. Furthermore, their own processing time is negligible. Such operators are, for example

```
rename:  stream(Tuple) x string -> stream(Tuple1)
count:   stream(Tuple) -> int
```

The *rename* operator just changes tuple schemas by appending the second argument to the attribute name, the tuple stream is just passed through. The *count* operator was discussed in Section 2.2. Similar operators are *sum*, *min*, *max*, *avg* which just aggregate over a stream. In such cases, the operator can simply return the progress information of its predecessor.

Operators Changing Tuple Sizes. Operators in this class change the tuple schema by either performing projection or adding derived attributes.

```
project:   stream(Tuple) x attrname+ -> stream(Tuple1)
extend:    stream(Tuple) x (newattrname x (Tuple -> Data))+-> stream(Tuple1)
groupby:   stream(Tuple) x attrname+ x
              (newattrname x (Tuple -> Data))+ -> stream(Tuple1)
```

The *project* operator gets a stream of tuples and a list of attribute names (`attrname+`) and returns a stream of projection tuples. For this operator it is obvious that one can compute new tuple size information based on the *Sizes* fields in the progress information of the predecessor. To avoid unnecessary computations, the attribute size vectors are allocated and filled only on the first progress query and then stored in the local data structure of the projection operator. On every subsequent progress query, a check is made whether the tuple size returned from the predecessor is still the one stored. Otherwise, attribute size vectors are recomputed.

The *extend* operator in addition to the stream argument gets a list of pairs. Each pair consists of an attribute name and an expression to be evaluated on the given tuple, returning a new attribute value. An example use would be

```
query Highways feed extend[InFog: intersection(.Route, fog)] consume
```

Here a new attribute *InFog* is computed, of a spatial data type *line*, describing the geometry of the highway lying inside a *fog* area. For this operator, one cannot determine the size of attribute values from the tuple schema but needs to observe it from the tuple flow. Here we proceed as follows. In the cold state, we assume as a default that the size of the new attribute is that of an integer.[7] As soon as tuples start to flow through the operator, for the first *s* tuples (currently *s* is set to 5), we get the size of each derived attribute from the new tuple and add it to a field in a temporary attribute size vector. When *s* tuples have been read, a stable state is assumed and the size information stored in the operator's local data structure is updated accordingly. From then on, progress queries will get the average size information observed on the first *s* tuples. The reason for observing only the first tuples is to avoid a large overhead in the normal processing.

An operator that is handled by similar strategies is *groupby*, which also computes derived attributes for each group of tuples. Its arguments are a stream of tuples ordered by grouping attributes, a list of attributes for grouping and a list of pairs (*attribute name*, *expression*) defining new attributes to be computed for each group.

Operators Changing Cardinality. Beyond filter, this is duplicate removal.

```
filter:  stream(Tuple) x (Tuple -> bool) -> stream(Tuple)
rdup:    stream(Tuple) -> stream(Tuple)
```

Filter was discussed in Section 2.2. The duplicate removal operator *rdup* works on a stream ordered lexicographically by all attributes (see the *sort* operator below) and returns for adjacent equal tuples only one instance.[8] In the cold state we do not have selectivity information from the optimizer available. Here we assume a reduction in the number of tuples to 90% of the incoming tuples. In the warm state, of course the observed ratio is used to determine the cardinality of the result stream. Other operators that change cardinalities and that can be handled by similar techniques are *groupby* and *extendstream*. The latter operator is similar to *extend* but the expression for the new attribute returns a stream of values. An extended copy of the input tuple is returned for each value of the attribute stream.

Sorting. There are two sort operators in SECONDO with essentially the same implementation.

```
sort:    stream(Tuple) -> stream(Tuple)
sortby:  stream(Tuple) x (attrname x dir)+ -> stream(Tuple)
```

The first sorts lexicographically by all attributes and is usually employed together with the *rdup* operator; the second sorts by the mentioned attributes (the `dir` value is from the set {`asc`, `desc`}). Progress estimation is done with the techniques of Section 2.2. Obviously, *sort*/*sortby* has a contribution to blocking time and progress for the sorting stage; this is modeled similarly as it was shown for the *product* operator in Section 2.3.

7. The rationale is that often integer attributes are derived. This is the current SECONDO implementation. One could improve estimation for the cold state by using the size information for the derived data type if it is of fixed size, or a type specific default, otherwise.
8. It is well known that duplicate removal by hashing is more efficient. The latter implementation is not available in the standard SECONDO system.This is in fact, because it is regularly added in a student exercise.

Join Operators.

```
product:        stream(Tuple1) x stream(Tuple2) -> stream(Tuple3)
symmjoin:       stream(Tuple1) x stream(Tuple2)
                  x (Tuple1 x Tuple2 -> bool) -> stream(Tuple3)
hashjoin:       stream(Tuple1) x stream(Tuple2)
                  x attrname x attrname x int -> stream(Tuple3)
mergejoin:      stream(Tuple1) x stream(Tuple2)
                  x attrname x attrname -> stream(Tuple3)
sortmergejoin: stream(Tuple1) x stream(Tuple2)
                  x attrname x attrname -> stream(Tuple3)
loopjoin:       stream(Tuple1) x (Tuple1 -> stream(Tuple2)) -> stream(Tuple3)
```

Product and *symmjoin* are discussed in Section 2.2. All join operators except *loopjoin* can be handled by techniques similar to those discussed for *symmjoin*. For example, they all use the same method to assign tuple sizes, use optimizer selectivities or defaults in the cold state, and the observed ratio in the warm state. *Hashjoin*[9] and *sortmergejoin* have blocking times; *mergejoin* is non-blocking, being applied to ordered streams.

The *loopjoin* operator is special as it has an opaque parameter function. It is supplied with each tuple of the outer stream. For each argument tuple it yields a stream of tuples. Those are joined by *loopjoin* with the outer tuple and returned. Usually it is used for index nested loop, but this is not required. *Loopjoin* calls the parameter function many times, once for each tuple of the outer stream. It also needs to ask the parameter function for its progress information. The question is how the parameter function, especially in case of an index access, can offer a precise estimate. For example, a progress query may reach the parameter function in the middle of evaluation for tuple #597 of the outer stream. The solution we have come up with is that operators like *exactmatch* (a B-tree access operator, see below) count how many times their *Open* message is called and remember how many tuples they have returned aggregated over all calls. As an estimate of the cardinality for the current call they can then return the average number of tuples returned over all previous calls. This technique works very well, as will be demonstrated in the experiments.

Stream Sources - Relation and Index Access.

```
feed:             rel(Tuple)    -> stream(Tuple)
exactmatch:       btree(Tuple) x rel(Tuple) x Data -> stream(Tuple)
range:            btree(Tuple) x rel(Tuple) x Data x Data -> stream(Tuple)
leftrange:        btree(Tuple) x rel(Tuple) x Data -> stream(Tuple)
rightrange:       btree(Tuple) x rel(Tuple) x Data -> stream(Tuple)
windowintersects: rtree(Tuple) x rel(Tuple) x Spatial -> stream(Tuple)
```

Feed has been discussed in Section 2.2. *Exactmatch*, *range*, *leftrange*, and *rightrange* are search operations on a B-tree (as a secondary index). Arguments are the index, the indexed relation and constants for search; these operators fetch tuples from the indexed relation and put them into the result stream.[10] In fact, these operators all share the same parameterized implementation. Similarly, *windowintersects* is a search operation on an R-tree, using the bounding box of the value of a spatial data type.

For index access operations, prediction of the total cardinality is not so easy. To achieve a simple implementation, we would like to avoid a deep analysis of the index traversal algorithm. Therefore in this case we check whether the selectivity obtained from the operator tree is the default selectivity. In this case, no good selectivity information is available and instead we use default values for the estimated cardinality. Otherwise the selectivity from the operator tree is used. For queries constructed by the opti-

9. The fifth argument for hashjoin is the number of buckets to be used.
10. There exist also variants returning streams of tuple identifiers; these are not discussed here.

mizer, these values should be fairly exact, as such index accesses happen on base relations and no correlations and propagated errors occur.

Progress is then measured by the number of tuples returned relative to the estimated cardinality. If the operation is finished before the estimated cardinality is reached, we set the estimated cardinality to the real cardinality. If the number of tuples returned exceeds the estimated cardinality based on the selectivity from the operator tree, we estimate a cardinality of 10% more tuples than have been returned already.

If the index is used in an index nested loop join, we have explained above how precise estimates can be given aggregating over previous calls, counting how many times the *Open* message was called.

5 Experiments

In this section, we show the results of some experiments to demonstrate operator-based progress estimation. Experiments are done on a PC with a 2.66 GHz CPU and 1 GB main memory, running Windows XP. Queries are performed on the standard TPC-H database [13] at scale factor 0.1, to avoid long waiting times. [11]

We consider the following queries, formulated as query plans. For each query, we show three graphs depicting estimated progress *P*, estimated cardinality *C* and the estimated remaining time. The latter is derived from progress (also in the SECONDO user interface) by the formula $T_{rest} = T_{elapsed} \cdot (1\text{-}P)/P$. This is compared to the real remaining time (shown as a straight line in the graphs) computed as T_{total} - $T_{elapsed}$. Note that in this paper we do not attempt to adapt progress estimation or the estimation of remaining time to varying system loads (as e.g. [6, 7] do). The formula above simply assumes that progress will be made in the future as quickly as it was made in the past. On the other hand, an extension to varying system loads is probably not difficult as one could relate future "speed of making progress" to the progress speed within a time window for the recent past that defines the current system load. But this is beyond the scope of this paper.

All quantities are shown relative to the elapsed time during execution of a query.

```
Query 1. LINEITEM feed filter[.lQUANTITY > 7] filter[.lSHIPDATE > theIn-
stant(1994,1,1)] count
```

The first query simply scans the *lineitem* relation evaluating two conditions. Results are shown in Figure 12. Note that operations are generally applied in Postfix notation in SECONDO.

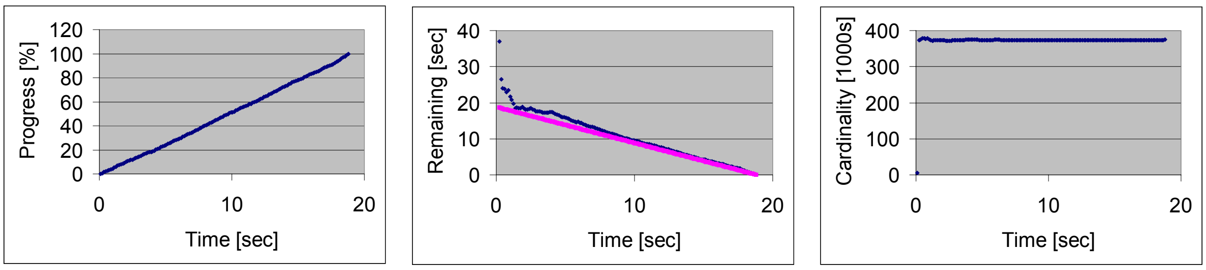

Figure 12: Query 1

11. Obviously, SECONDO as a research prototype is a bit slower than commercial systems.

One can observe that progress develops smoothly and the estimate of the remaining time is quite close. Also the cardinality estimate is quickly very precise. The real cardinality is 374232. (In all following queries the estimate at the end is the right one, so we will not mention the actual cardinalities any more.) Note that this is a query plan without any use of optimizer estimates.

```
Query 2. LINEITEM feed filter[.lQUANTITY > 7] consume feed filter[.lSHIPDATE >
theInstant(1994,1,1)] count
```

This is essentially the same query but we have put a blocking operator into the middle of the query plan. See Figure 13.

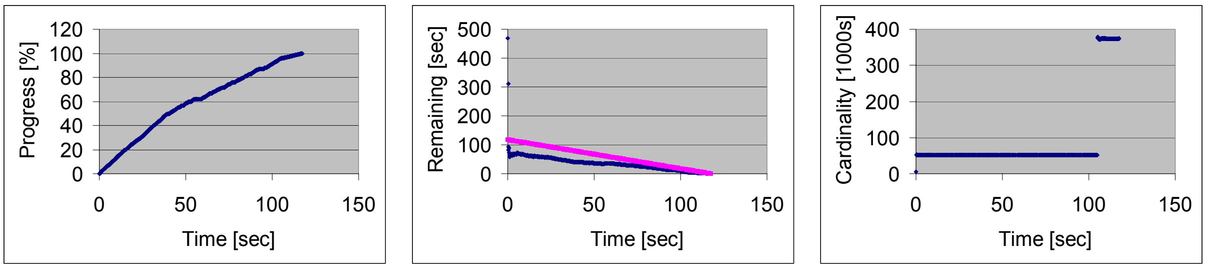

Figure 13: Query 2

Due to the blocking operator, nothing is known about the selectivity of the second condition until the second *filter* operator gets into the warm state. As a result, cardinality estimation is wrong until then and the estimate of the remaining time is considerably less precise. The progress curve looks still relatively straight because the majority of the work belongs to the first stage (writing tuples in consume is expensive).

Next, we consider application of the *head* operator to both the non-blocking and the blocking version of this query.

```
Query 3. LINEITEM feed filter[.lQUANTITY > 7] filter[.lSHIPDATE > theIn-
stant(1994,1,1)] head[50000] count

Query 4. LINEITEM feed filter[.lQUANTITY > 7] consume feed filter[.lSHIPDATE >
theInstant(1994,1,1)] {0.710789, 0.1875} head[50000] count
```

In query 4, we have additionally inserted selectivity information obtained from the optimizer. Results are shown in Figures 14 and 15.

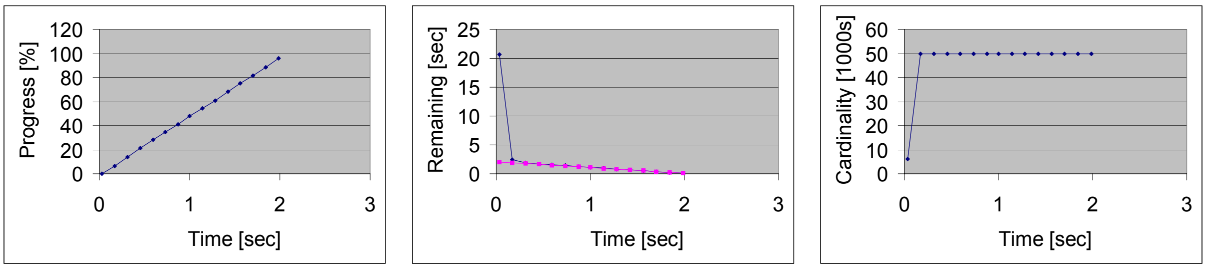

Figure 14: Query 3

Query 3 has only about 20 progress measurements as the query runs only for two seconds. We have connected them by a curve for better visibility. One can observe that progress estimation with the head operator works fine, regardless of whether there is a blocking operator in the query plan. Hence our strategy with propagating blocking time and blocking progress is confirmed. For query 4 in addition estimation has become precise due to the available selectivity information from the optimizer.

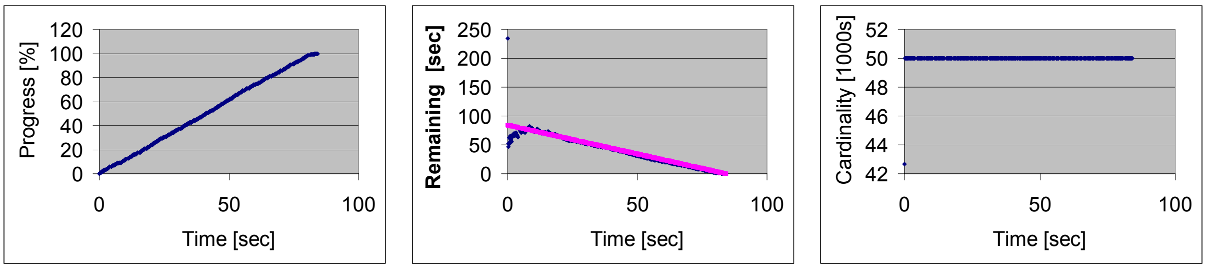

Figure 15: Query 4

```
Query 5. select count(*) from [lineitem, orders] where [ototalprice > 300000,
oorderkey = lorderkey]

ORDERS  feed project[oORDERKEY, oTOTALPRICE]
  filter[(.oTOTALPRICE > 300000)] {0.034965, 0.1095}
  loopjoin[lLINEITEM_lORDERKEY LINEITEM  exactmatch[.oORDERKEY]
    project[lORDERKEY] ] {7.992e-006, 0.016628}
  count
```

Query 5 is meant to illustrate the *loopjoin* estimation discussed in Section 4. Here we have run the query through the optimizer. See Figure 16. One can observe that the progress by *loopjoin* is very smooth and the estimates of cardinality and remaining time quickly become very precise (note the scale in the cardinality estimate).

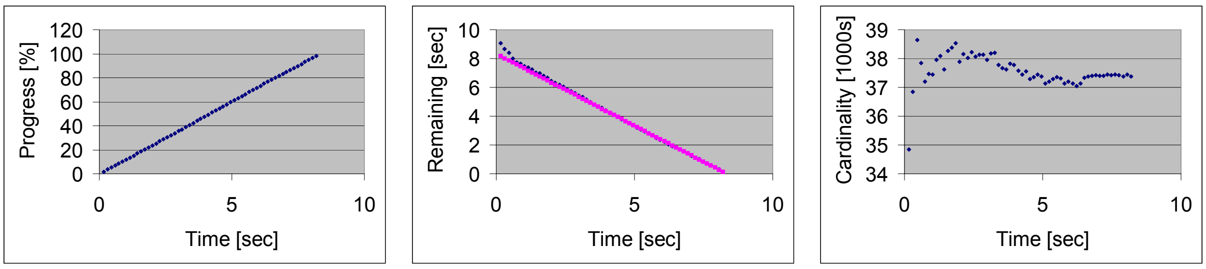

Figure 16: Query 5

Query 6 shows nested hashjoins following an example from [6]; it is Query 2 in that paper.

```
Query 6. select [ccustkey, cacctbal, oorderkey, ototalprice, ldiscount, lex-
tendedprice] from [customer, orders, lineitem] where [ccustkey = ocustkey,
oorderkey = lorderkey, abs(int2real(lpartkey)) > 0]

CUSTOMER feed project[cACCTBAL, cCUSTKEY]
ORDERS feed project[oCUSTKEY, oORDERKEY, oTOTALPRICE]
  hashjoin[cCUSTKEY, oCUSTKEY, 99997] {5.5944e-005, 0.013308}
LINEITEM  feed project[lDISCOUNT, lEXTENDEDPRICE, lORDERKEY, lPARTKEY]
  filter[(abs(int2real(.lPARTKEY)) > 0)] {0.999, 0.172}
  hashjoin[oORDERKEY, lORDERKEY, 99997] {7.992e-006, 0.016628}
  project[cCUSTKEY, cACCTBAL, oORDER KEY, oTOTALPRICE, lDISCOUNT,
    lEXTENDEDPRICE]
  consume
```

The second part is the plan generated by the optimizer including selectivity and predicate cost annotations. Results are shown in Figure 17.

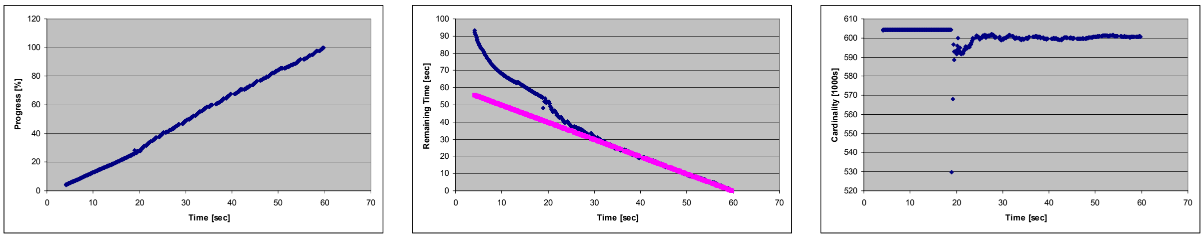

Figure 17: Query 6

One can observe that cardinality estimation first relies on the optimizer estimate which is fairly good. As soon as the second *hashjoin* gets into the warm state, selectivity estimation is refined. The first estimates are a bit off, but this quickly stabilizes.

```
Query 7.
CUSTOMER feed project[cACCTBAL, cCUSTKEY]
ORDERS feed project[oCUSTKEY, oORDERKEY, oTOTALPRICE]
  hashjoin[cCUSTKEY, oCUSTKEY, 99997]
LINEITEM feed project[lDISCOUNT, lEXTENDEDPRICE, lORDERKEY, lPARTKEY]
  filter[(abs(int2real(.lPARTKEY)) > 0)]
  hashjoin[oORDERKEY, lORDERKEY, 99997]
  project[cCUSTKEY, cACCTBAL, oORDERKEY, oTOTALPRICE, lDISCOUNT,
    lEXTENDEDPRICE]
  consume
```

This is actually the same query and the same plan as in query 6, but we have removed all optimizer estimates. Results are shown in Figure 18.

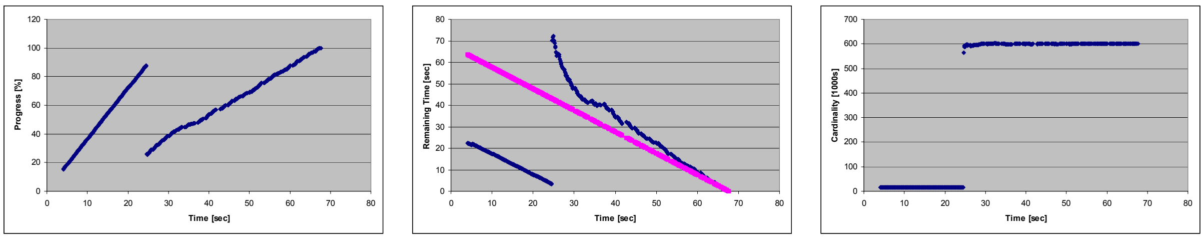

Figure 18: Query 7

Comparing to Figure 17, one can see that the lack of optimizer information is disastrous. As long as the second hashjoin is in the cold state, no reasonable estimate of the total work can be made. Hence progress is vastly overestimated in the first stage. After about 25 seconds, the second hashjoin gets into the warm state and then estimations have to be sharply revised. This experiment confirms the expected: optimizer annotations in query plans are indispensable for queries with blocking operators.

Finally, Figure 19 shows the progress graphs for the four TPC-H queries 1, 3, 5, and 10. These are all the TPC-H queries that SECONDO can evaluate as its optimizer does not support subqueries.[12] The graph for query 1 shows some steps; this is due to the fact that the query computes aggregates over four large groups and progress is not increased while the group-by operator scans its buffer. The graph for query 3 is not quite satisfactory; presumably the paging behaviour of a join algorithm is not modelled

12. Note that this is just a gap in the optimizer. Basically, subqueries can be unnested into joins or translated less efficiently into nested loops. The SECONDO execution system could handle these translations and progress estimation would work for them. We are currently working on implementing subqueries in the optimizer.

precisely enough. Nevertheless the graphs demonstrate that the progress estimation is quite usable, also for complex queries.

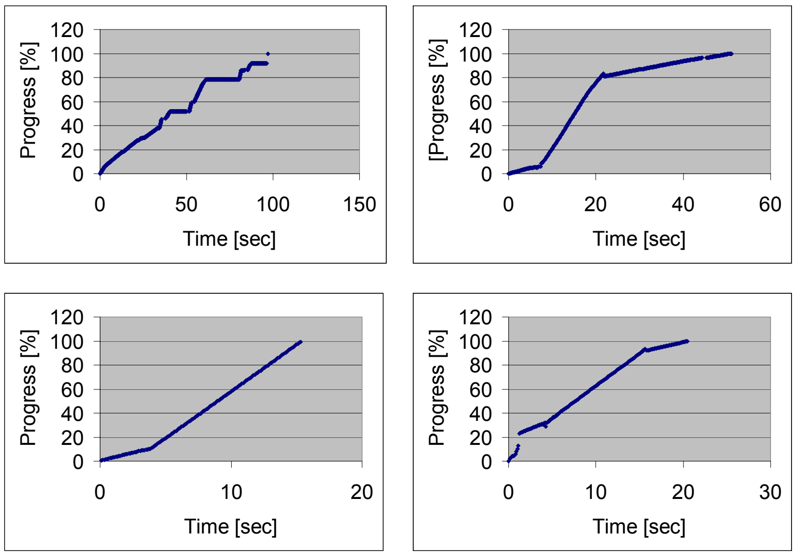

Figure 19: Progress Graphs for TPC-H Queries 1, 3, 5, 7 (top left, right, bottom left, right)

6 Discussion

In this section we discuss some questions that have not been addressed so far, namely overhead, “ad-hoc” constants and thresholds for switching to the warm state, and dealing with non-random orders.

6.1 Overhead

The following kinds of overhead for progress estimation arise in our implementation:

(i) Watching the time in the *Eval* function to check whether a progress query should be sent.
(ii) Evaluating a progress query and adjusting the progress display. This happens roughly ten times per second.
(iii) Maintaining counters in the implementation of query processing operators. Essentially for each tuple read or returned, a counter is incremented.
(iv) For very few operators (*extend* and *groupby*) observing tuple sizes for the first few tuples of a tuple stream (currently 5 tuples, see Section 4).

The cost for item (i) is extremely small, as can be seen above, and we have not been able to measure it. Similarly, the cost for incrementing counters in tuple processing (item (iii)) is clearly negligible compared to other code that is executed.

For item (iv), the cost will usually be very small, as only the first few tuples of each stream create this overhead. Nevertheless, situations are conceivable, for example in processing group-by queries, when a derived attribute of unknown size is computed for each tuple of a group, and all groups are quite small so that a large fraction of the tuples processed will have the overhead of measuring attribute sizes. In such cases a measurable overhead may exist. However, these cases are very rare and we have not yet checked this experimentally.

Hence what is still an open question is the cost of processing a progress query (item (ii)). Since here larger sections of code are executed, one might expect a tangible overhead. We have evaluated this experimentally.

In the experiment, a script was run that executes the four TPC-H queries 1, 3, 5, and 10 (see also Figure 19). These are sizeable queries whose query plans involve a large number of operations. The *Eval* function of the query processor was modified for the experiment to trigger a loop of 1000 progress queries after 5 seconds of query evaluation. Since each progress query creates an entry in a protocol file which includes the clock time, one can measure rather precisely the cost for 1000 progress queries. The results are shown in Table 1.

Query	start time loop [ms]	end time loop [ms]	time for 1000 progress queries [ms]	per query [ms]
Query 1	5062	5109	47	0.047
Query 3	5016	5062	46	0.046
Query 5	5032	5079	47	0.047
Query 10	5047	5094	47	0.047

Table 1: Evaluating 1000 progress queries

For example, the protocol file contains 1000 lines with equal progress value shortly after time 5000 for query 1 and the first line has clock time 5062, the last 5109. Hence the time per progress query is about 47 microseconds for fairly complex queries. Times for queries 1 through 4 are almost equal because system clock times are not incremented continuously but only in increments of about 15 to 16 ms. We conclude that the overhead for processing ten progress queries per second is about 0.5 ms, hence 0.05%.

With respect to overhead, previous papers have mentioned that it is low [3, p. 804 bottom] or negligible [3, p. 809 top]. There are no experiments about this. Reference [6, Section 5, first par.] says that progress indicators could be updated every ten seconds with less than 1% overhead. This seems to imply that performing a progress evaluation takes about 0.1 seconds. Compared to 0.047 milliseconds, the overhead appears to be larger by a factor of about 2000.

We do not suggest that overhead is a problem in the other approaches, but definitely in SECONDO it is very small.

6.2 Default Values and Threshold for the Warm State

There are two kinds of ad-hoc constants used in the approach:

- *qp.Sel* - the default selectivity of a predicate and *qp.PredCost* - the default cost of a predicate, with *qp.Sel* = 0.1 and *qp.PredCost* = 0.1.
- The threshold for the number of tuples returned by filter and join operators for switching into the warm state, i.e. the number of positive predicate evaluations. Here the threshold value 50 was mentioned.

Default Values. For the first kind of constants, please note that they are only used when queries are posed at the executable level, i.e. entered into the system as query plans without using the optimizer. This is a mode of operation that is not at all supported by the systems described in previous work (see Section 7) which always start from optimizer estimates.

The constant 0.1 for the selectivity of an arbitrary predicate is clearly ad-hoc, but what can one do without query optimization and when selectivity has not yet been observed? On the other hand, the constant 0.1 [ms] for predicate cost is a fairly reasonable assumption that holds for simple standard predicates in SECONDO.

Threshold for the Warm State. Obviously, selectivity estimation gets the more reliable the more evaluations have been observed. So why is 50 a reasonable value; could it be 5 or should it be 1000?

Our goal in this subsection is to analyse the reliability of the estimate depending on the threshold value. To make a mathematical analysis possible, we first make some (idealized) assumptions.

- A stream of tuples passing through a filter operator appears in random order with respect to the selection predicate.
- The pairs of tuples examined in a join operator appear in random order with respect to the join predicate.
- For each tuple or pair of tuples examined, the probability that the predicate is fulfilled is the same and equal to the selectivity of the predicate.

These assumptions will not always hold, but they help us to get a handle on reasonable sizes of the threshold. In the next subsection we discuss how to deal with non-random orders.

If the assumptions hold, the number of successes for the sequence of predicate evaluations can be modelled by a binomial random variable. That is, the probability that after *n* trials (tuples, pairs of tuples), each with probability *p* of being a success, we have *k* successes (returned tuples), is:

$$\binom{n}{k} p^k (1-p)^{n-k}$$

It is well known that the binomial distribution can be approximated by a normal distribution, and that this approximation is quite good when $np(1-p) \geq 10$. This is useful when *n* is large.

Now suppose *w* is our threshold for the warm state and we have seen *w* successes after *n* experiments (i.e. have returned *w* tuples after *n* tests). We will then estimate the selectivity to be $p' = w/n$. What is the probability that the estimated selectivity p' is within a factor *e* of the true selectivity *p*?

We argue as follows. We will consider the two cases (i) $p' < p$ and the error is greater than factor *e*, and (ii) $p' > p$ and the error is greater than factor *e*.

Case (i). Suppose $p' < p$ and the error is greater than factor *e*, hence $p' < (1 - e)\,p$. The unknown true selectivity *p* is anywhere in the interval $[p'/(1 - e), 1]$. Now consider the case that the true selectivity *p* is as close as possible to the estimated selectivity p', hence let $p = p'/(1 - e)$. For this selectivity, the probability *c* that the outcome of an experiment is in fact within the error range *e* is given by the cumulative distribution function of the normal distribution within the range $[(1 - e)\,p, (1 + e)\,p]$.

Figure 20 illustrates the case that for $n = 1000$, $w = 40$ the outcome of our "experiment" is $e = 1/3$ off the expected value of 60 for a given true selectivity of 0.06. The area under the curve within the range [40, 60] defines the probability that the error for such an outcome of the experiment is less than 1/3.

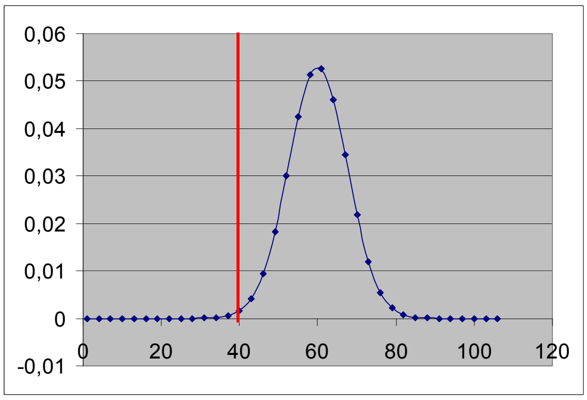

Figure 20: Normal distribution corresponding to parameters $p = 0.06$, $n = 1000$

If the true selectivity p is larger than $p' / (1 - e)$, then the area under the curve gets even larger, as the normal distribution curve is shifted to the right (this seems obvious but is not proved here). Therefore c is a lower bound for error e and the case $p' < p$.

Case (ii). By similar reasoning we can obtain a lower bound c' for error e and the case $p < p'$.

Finally $c'' = \min\{c, c'\}$ is a lower bound on the probability that the error is within factor e of the true selectivity p.

Of course, one can argue about what kind of precision is still useful for progress estimation. In any case, with $w = 50$ selectivity estimates are fairly reliable if the assumption holds that tuples arrive in random order.

We have evaluated c'' for thresholds $w = 10, 20, 50$, and 200, errors ranging from 10% to 50% and various numbers n of tuples checked. The results for $w = 10$ and $w = 50$ are shown in Tables 2 and 3.

Error *e*	*n* = 200	500	1000	10000	500000
10%	0.24	0.24	0.24	0.24	0.24
20%	0.44	0.44	0.44	0.44	0.44
30%	0.60	0.60	0.60	0.59	0.59
40%	0.72	0.72	0.72	0.72	0.71
50%	0.81	0.81	0.80	0.80	0.80

Table 2: Threshold $w = 10$

Error *e*	*n* = 200	500	1000	10000	500000
10%	0.56	0.52	0.51	0.50	0.50
20%	0.85	0.82	0.81	0.80	0.80
30%	0.96	0.95	0.94	0.94	0.94
40%	0.99	0.99	0.99	0.98	0.98
50%	1.00	1.00	1.00	1.00	1.00

Table 3: Threshold $w = 50$

One can observe that for a threshold $w = 10$ the probability is only about 80% that the error in the selectivity estimation is less than 50%. For threshold $w = 50$, with at least 94% probability we can guarantee that the error is less than 30%.

6.3 Dealing With Non-Random Orders

As we have just seen, adaptive selectivity estimation relies on the assumption that tuples are processed in random order with respect to the predicate evaluated.

Note that the assumption is NOT that tuple streams occur in random order, but that their order is unrelated to the predicate evaluated. So one has to take special care of the case that a predicate is evaluated on a tuple stream that is related to the order of that stream.

One can easily see what should not happen: A relation R is stored ordered on attribute X and then a range predicate of the form $X < a$, $X > a$, or $a < X < b$ is evaluated by scanning the relation, that is:

```
    R feed filter[.X < a] ...
```

For example, for predicate $X < a$ the selectivity would be estimated to be about 1.0 after switching into warm state until a is reached; then it would decrease to a possibly quite small value, if a is small.

This can be generalized to relations stored clustered in any way, for example, organizing spatial data in a (primary) R-tree, and then asking any kind of range query.

Ordered tuple streams occur mainly for the following reasons:

- A relation is stored in primary key order, or otherwise clustered by some attribute, and then scanned.
- A tuple stream is created through an index access and has the order of that index.
- Intermediate operations in query processing, e.g. sortmergejoin, sorting, create the order.

We discuss two possible strategies to avoid wrong selectivity estimates due to order:

- Keeping track of orders
- Checking for random order

Keeping Track of Orders. The first strategy relies on the fact that query optimizers generally are aware of the orders of base relations and keep track of the orders of the tuple streams in the query plans they generate. The query plans constructed are then modified according to the following rules:

- If a base relation is stored in some order (e.g. organized by a B-tree or R-tree), then a predicate related to that order is never evaluated by scanning the relation. Instead, the appropriate access operation such as a range query on the B-tree or R-tree is used. In fact, most optimizers will do this anyway.
- If an order related predicate is evaluated on an ordered stream, the filter or join operator is informed by a suitable mechanism not to go into warm state, but instead to stay with the optimizer estimate. (For example, the selectivity estimate used in the annotation of the query may be marked in some way.)

Essentially this switches off the adaptive behaviour of that operator when it would be wrong.

Checking for Random Order. The idea of the second strategy is to check in the implementation of an adaptive operator whether the tuple stream arriving appears to be in random order.

Consider a sequence of predicate evaluations that have occurred in the *filter* operator at the time when it is supposed to switch to the warm state (see Figure 21). Here the threshold is $w = 5$, i.e., 5 positive evaluations are needed to switch to the warm state.

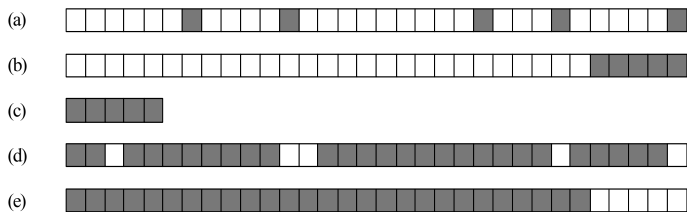

Figure 21: Sequences of predicate evaluations (black = *true*, white = *false*)

Intuititively, for an input stream in random order we would expect a pattern like (a). In contrast, pattern (b) could correspond to a predicate $a < X < b$ where value a has just been passed.

For pattern (c), one cannot decide whether it is a random pattern for a very high selectivity, or a range query of the form $X < a$ which will switch to *false* some time later. For this reason, we now extend the threshold condition to require that both the number of positive and the number of negative evaluations must be larger than the threshold w. Under this assumption, the transition to warm state is reached in patterns (d) and (e) since enough evaluations with outcome *false* have been observed. Patterns (d) and (e) are then symmetric to (a) and (b): pattern (d) appears to show random order and pattern (*e*) exhibits some order.

Different criteria can be used to decide whether a sequence of outcomes appears to confirm random order. If only range predicates on totally ordered streams need to be handled, one could simply check whether all positive evaluations follow all negative evaluations, as in pattern (b), or the symmetric case as in (e). This criterion is too strict, if also predicates like range queries on spatial data are considered, where the base relation is stored in *z*-order, for example. Hence we suggest a more general criterion based on the lengths of intervals of the same result. For example, pattern (a) has “white” intervals of lengths 6, 4, 9, 3, 5. For a random order, the expected length of such an interval is n/w, that is, $32/5 = 6.4$. Pattern (b) has a white interval of length 27.

Let *pos*, *neg* denote the numbers of positive and negative evaluations, respectively, and let $pos < neg$, $pos = w$, $neg = n-w$. Let *maxneg* denote the length of the longest continuous interval of negative evaluations. Also assume $w \geq 20$. A suitable criterion might be:

The sequence is supposed to be in random order $:\Leftrightarrow$ $maxneg < n/2$.

Similarly for $pos \geq neg$, the criterion would be $maxpos < n/2$.

Essentially this means that if we observe a continuous subsequence of more than $n/2$ negative evaluations, given that the expected length of such a sequence is $n/w < n/20$, we have enough suspicions that this is not a random order. In this case the operator will not go into warm state and instead continue to rely on the optimizer estimate.

Keeping track of the maximal length of positive or negative intervals is easy to implement in constant space and adds only negligible overhead.

In summary, both strategies essentially switch off adaptive selectivity estimation for operators that evaluate predicates related to the order of the processed stream of tuples, and so prevent wrong estimates. The second strategy may be easier to implement as it does not require changes to the optimizer.

6.4 Pipelining

Previous work (especially [2, 3]) has argued that observation of the behaviour only of driver nodes in a pipeline for estimating the progress of the whole pipeline is more robust against errors in selectivity estimation than observing each operator node individually. This is true in principle. Nevertheless, there are two answers to this:

(i) If the advice of the previous two subsections is followed and the optimizer does a reasonable job, then usually the intermediate selectivity estimations in a chain of operators will be quite precise. This is also confirmed by our experiments.
(ii) It is possible to some extent to support pipelining in our operator-based approach, still without ever analyzing query plans globally.

To see (ii) consider a chain of operators $op_1\ op_2\ ...\ op_n$ working in a pipeline with op_1 as a driver node. For example, op_1 may be *R feed*. Pipelined progress estimation means that

$$Progress(op_{i+1}) = Progress(op_i), \text{ for } 1 < i \leq n.$$

Hence the *feed* operator with its very robust progress measure $P_{feed} = m/C$ determines the progress of all operators in the pipeline (m the number of tuples read, C the total cardinality). This means that for operators *op* like *filter*, *project*, *consume*, *loopjoin*, etc. we should define

$$P_{op} = P_1$$

Instead, in Section 2.2 we have defined progress for such operators in the form:

$$P_{op} = (P_1T_1 + mX) / (T_1 + CX) \qquad (*)$$

Here X is the cost of processing a tuple. When do the two definitions agree? If we set $P_1 = (P_1T_1 + mX) / (T_1 + CX)$ we can derive

$$P_1 = m/C$$

Hence, if the driver node of a pipeline has a progress measure m/C and it is followed by a chain of operators with definitions of the form (*), then indeed all these operators have progress m/C and we can ignore intermediate cardinality estimates.

The simple formula $P_{op} = P_1$ is not valid, however, if the progress of the predecessor, i.e. P_1, is different from m/C. This is the case when blocking operators occur before this pipeline.

Fortunately, with the techniques of Section 2.3, we know whether there are blocking operators in a query plan; this is the case when the blocking time of the predecessor is different from 0. As a consequence, we have modified[13] the progress formulas for suitable operators like e.g. *filter*, *project*, *consume*, *loopjoin*, to the form:

$$P_{op} = \begin{cases} P_1 & \text{if } BT_1 \approx 0 \\ (P_1T_1 + mX)/T_1 + CX & \text{otherwise} \end{cases}$$

Hence any initial pipelines in a query plan before blocking operators enjoy the robustness of pipelining. First experiments designed to create very wrong selectivity estimates (without using the safeguards of Section 6.3) indeed show a better behaviour of the pipelined version. A detailed experimental comparison is left to future work.

13. The current SECONDO implementation allows one optionally to select this "pipelined" mode or the formulas presented before.

7 Related Work

The approaches of the two groups [3, 2], and [6, 7, 8] are somewhat similar. We first discuss the initial papers [3, 6]. Both decompose the operator tree for the query into disjoint subtrees such that within each subtree operators are non-blocking and work in a pipeline [3] (called segment in [6]). Both approaches identify entry points of the pipeline as points to be observed (called driver nodes in [3], dominant inputs in [6]). As a measure of progress, [3] use the number of tuples returned by either all operators or just the driver nodes; [6] count numbers of bytes corresponding to full pages that are read or written at the boundaries of segments. Both assume that the cost per tuple or page in different pipelines (segments) is the same.

Both approaches start from the given optimizer estimates of intermediate cardinalities. [6] refine these estimates based on observed selectivities using a gradual transition from the optimizer estimate to the observed selectivities. [3] are more conservative and maintain upper and lower bounds on cardinalities; only if by algebraic properties of the operators it is certain that the optimizer's estimate is wrong (i.e. lies outside the bounds), the estimate is corrected.

The follow-up paper [7] tries to increase the scope of the techniques, e.g. by handling nested queries, as well as the accuracy. The authors observe that the granularity of segments is too large to make precise predictions in certain cases and come up with a refined definition such that a segment contains no more than one join operator. In the next step, a segment is redefined to also contain only one set operator (like set difference on ordered streams). The approach of [6, 7] is further extended in [8] to consider multiple queries that are running concurrently or even predicted to arrive in the future.

In [2] the authors focus on deriving worst case guarantees, considering the problem from a theoretical point of view. They show that in the worst case only trivial progress estimations can be given. But they also identify "good" scenarios where rather tight bounds can be provided.

Mishra and Koudas [10] improve cardinality estimations for joins and pipelines of joins by maintaining histograms for intermediate results. They also develop techniques to precisely estimate the number of groups in groupby operators.

Progress estimation is more generally related to work using feedback from query execution. This includes reoptimization techniques [5, 9, 1], methods to improve statistics for future queries, e.g. [12], or online aggregation [4].

8 Conclusions

We have shown that a strictly modular, operator based query progress estimation is possible. Major advantages over previous work are the following:

- From a software engineering point of view, the approach is far more modular as each operator can be considered independently. Adding further query processing operators to a system or to progress estimation is much easier.
- A precise cost modeling for each operator is possible, as in query optimization, so that the cost for not yet active operators (pipelines) can be estimated precisely. Hence there is a potential for a more precise progress estimation.
- Because the technique is integrated deeply into query processing, the overhead is extremely low. For the progress queries themselves we have shown it to be less than 0.1 %.

Future work includes a more precise cost modeling for complex queries, especially to model more exactly operators that need disk I/O such as sorting and sort-mergejoin. So far, this was modeled only for hashjoin somewhat precisely. Furthermore, we now plan to extend progress estimation to the non-standard applications in SECONDO, dealing with spatial join and expensive predicates, for example.

Acknowledgments

I am grateful to Werner Detemple, Markus Spiekermann, and Thomas Behr for their help in implementing progress estimation in SECONDO.

References

[1] S. Babu, P. Bizarro, and D. DeWitt, Proactive Re-Optimization. Proc. ACM SIGMOD 2005, 107-118.
[2] S. Chaudhuri, R. Kaushik, and R. Ramamurthy, When Can We Trust Progress Estimators for SQL Queries? Proc. ACM SIGMOD 2005, 575-586.
[3] S. Chaudhuri, V. Narasayya, and R. Ramamurthy, Estimating Progress of Long Running SQL Queries. Proc. ACM SIGMOD 2004, 803-814.
[4] J.M. Hellerstein, P.J. Haas, and H.J. Wang, Online Aggregation. Proc. ACM SIGMOD 1997, 171-182.
[5] N. Kabra and D. DeWitt, Efficient Mid Query Reoptimization of Sub-Optimal Query Execution Plans. Proc. ACM SIGMOD Conf. 1998, 106-117.
[6] G. Luo, J.F. Naughton, C.J. Ellmann, and M.W. Watzke, Toward a Progress Indicator for Database Queries. Proc. ACM SIGMOD 2004, 791-802.
[7] G. Luo, J.F. Naughton, C.J. Ellmann, and M.W. Watzke, Increasing the Accuracy and Coverage of SQL Progress Indicators. Proc. ICDE 2005, 853-864.
[8] G. Luo, J.F. Naughton, and P.S. Yu, Multi-query SQL Progress Indicators. Proc. EDBT 2006, 921-941.
[9] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic, Robust Query Processing Through Progressive Optimization. Proc. ACM SIGMOD 2004, 659-670.
[10] C. Mishra and N. Koudas, A Lightweight Online Framework for Query Progress Indicators. ICDE 2007.
[11] SECONDO Website. http://www.informatik.fernuni-hagen.de/import/pi4/Secondo.html/
[12] M. Stillger, G. Lohman, V. Markl, and M. Kandil, LEO: DB2's Learning Optimizer. Proc. VLDB 2001, 19-28.
[13] TPC Benchmark H. Decision Support. http://www.tpc.org.