An Overview of Eddies and STAIRs

Magnus Ågren

1 Problems with Static Query Plans

In databases where different resources have much changing characteristics, static query plans is often not a good solution. The problem is that, while a particular query plan might be a good alternative with some initial characteristics of the database, it may be a bad alternative when the characteristics changes. Consider, for example, joining the tables R, S and T, and assume first that R and S contain 1000 rows and T contains 10 rows. Clearly, it is best to not use the join ordering $(R \bowtie S) \bowtie T$ but to choose any other ordering such as $(T \bowtie R) \bowtie S$. However, if the size of T drastically increases to $1,000,000$ rows, this is not a good choice anymore.

In this overview we present the ideas of [1, 2] which suggests two adaptive query operators called Eddies and STAIRs. Using these operators instead of traditional ones, the query plans need not be fixed, but can be changed on a tuple-by-tuple basis. In the following, Section 2 starts by presenting the Eddy operator and discuss some of its shortcomings. After that, Section 3 presents the STAIR operator and how it can be used to overcome these shortcomings. Finally, Section 4 concludes the overview. For in-depth information about Eddies and STAIRs see [1, 2].

2 Eddies: Continuously Adaptive Query Processing

The *eddy* [1] was introduced to limit the problems with static query plans in a database with changing characteristics. An eddy has several inputs and one output, and functions as an adaptive router between several operators. Each input corresponds to a relation and an incoming tuple is routed to all the eddy's operators, in some order, before it is sent to the output. The eddy is an adaptive router since, for an active query, it may change the routing order of a tuple among its operators. By doing this, the eddy can adapt to changing characteristics of the database and be more efficient than a static query plan on the same operators. An example (from [1]) of an eddy is shown in Figure 1, with three input relations R, S and T, and four operators.

In general, an eddy consists of the following:

- A number of n input relations $R_1, \ldots, R_n$.

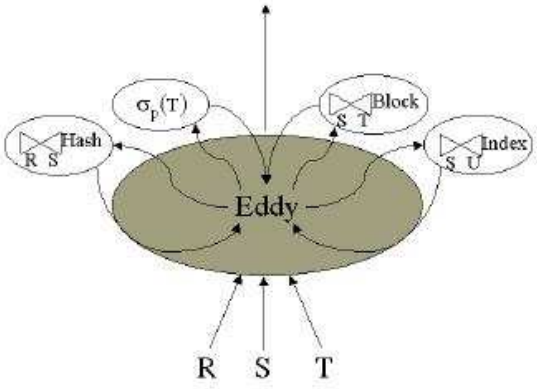

Figure 1: [1] An eddy with input relations R, S and T, and four operators.

- One output relation.
- A number of m unary or binary operators $O_1, \ldots, O_m$, each operator O_i having one or two inputs and one output.
- A priority queue for buffering unfinished tuples.

Each tuple t that is accepted on one of the eddy's input relations is assigned a $ready_t$ bit-string and a $done_t$ bit-string, both of length m, corresponding to the number of operators of the eddy. Bit $ready_t[i]$ is on (denoted by 1) if and only if operator O_i may accept t. Bit $done_t[i]$ is on if and only if relation O_i has accepted and returned t back to the eddy. By assigning different *ready* bit-strings to different tuples, different routings can be realised. When all $done_t$ bits are on, the tuple t is sent to the eddy's output. If t has some *done* bits off (denoted by 0), it is stored in the priority queue of the eddy, waiting for its turn to be accepted by its next operators, according to the on-bits of $ready_t$.

To illustrate this, consider the example shown in Figure 2. It displays an eddy with three incoming relations R, S and T, and two joins $R \bowtie S$ and $S \bowtie T$. In Figure 2(a), we see that the eddy's priority queue is filled with (in order of priority) the tuples s, r and t, where each tuple is of its corresponding capital-letter relation. We let $R \bowtie S$ be operator O_1 and $S \bowtie T$ be operator O_2. The *ready* and *done* bit-strings for s, r, and t are then respectively:

$$\begin{aligned} ready_s &= 11 \\ done_s &= 00 \\ ready_r &= 10 \\ done_r &= 00 \\ ready_t &= 01 \\ done_t &= 00 \end{aligned}$$

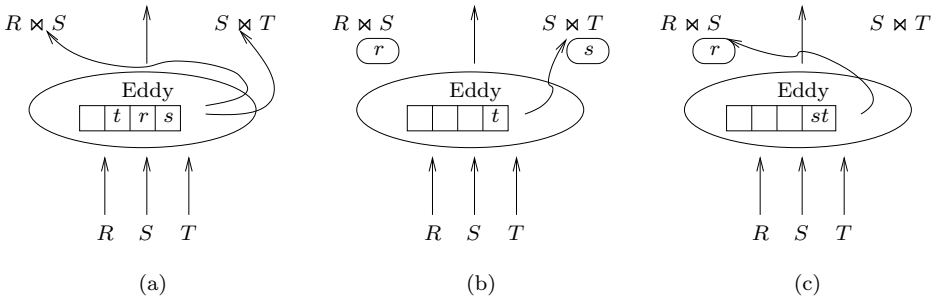

Figure 2: Three different states of an eddy with three input relations R, S and T, and two joins $R \bowtie S$ and $S \bowtie T$.

As we can see, there is a choice for tuple s to go first to either $R \bowtie S$ or $S \bowtie T$, as indicated by the $ready_s$ bit-string as well as the two arrows from the priority queue in Figure 2(a). This choice corresponds to the join orderings $(R \bowtie S) \bowtie T$ and $R \bowtie (S \bowtie T)$, respectively. Choosing $S \bowtie T$ for s, we see in Figure 2(b) the eddy where s has been accepted by $S \bowtie T$, and where r has been accepted by $R \bowtie S$ (the only choice for r). Figure 2(c) then shows the eddy where t has been accepted by $S \bowtie T$ (the only choice for t) and the joined tuple st has been returned to the eddy's priority queue. The bit-strings for st are then set to respectively $ready_{st} = 10$ and $done_{st} = 01$. Hence, the only choice for st is $R \bowtie S$, after which has accepted st, the joined tuple rst can be sent to the eddy's output.

By making different choices for routing tuples in the eddy, such as for the tuple s above, we can make the eddy route different tuples in different ways. Various heuristics for doing this can be imagined, some of which are evaluated in [1].

Unfortunately, there are still problems with the eddy operator which means that it may make current decisions resulting in poor performance in the future. This is partly because the operators within the eddy may buffer a large number of tuples, forcing the eddy to stick to certain routing decisions for a long time, even though it is clear that some other routing decisions would be better. Consider, for example, the eddy in Figure 2, and assume that initially the size of R and S are much larger than T. Assume also that the eddy initially routed the tuples of S to the join $S \bowtie T$ and that a large number of S tuples was buffered in $S \bowtie T$, waiting for T tuples to arrive. The join ordering will then be $R \bowtie (S \bowtie T)$ for all these tuples. Assume now that T grows to become much larger than R and S, meaning that the join ordering $R \bowtie (S \bowtie T)$ is not a good choice anymore. Since there are a large number of S tuples buffered in $S \bowtie T$, the eddy must stick to this ordering for some time resulting in poor performance.

3 STAIRs: Lifting the Burden of History from Adaptive Query Processing

The problem with eddies mentioned in the end of the previous section is handled by what is called a *STAIR* operator [2], which we will now describe. This operator uses ideas developed for the *SteM* operator [3], partly by the same authors. The basic idea behind the STAIR operator is to (from [2]):

> *"expose the state in the (eddy's) operators to the eddy, and allow the eddy to manipulate this state"*

In this way, when the eddy ends up in a state forcing it to make bad routing decisions, the eddy may manipulate its state in order to (continue to) make good routing decisions.

A join operator over two relations within the eddy is replaced by a pair of STAIR operators, associated with each of the relations of the join. The basic functionality of a STAIR operator is achieved by the following operations:

- $insert(R.a, t)$ stores the tuple t in the STAIR $R.a$ where a is the join attribute.
- $probe(R.a, v)$ returns all tuples r in the STAIR $R.a$ where $r.a = v$.

By using these operations it is now easy to mimic a join operator by a pair of STAIRs. For example, assume that the join $R \bowtie S$ is replaced by the two STAIR operators $R.a$ and $S.a$ and consider joining r from R with s from S. Using the join $R \bowtie S$ this would be done as follows (assuming that r arrives before s):

- Add r to the buffer of the join operator.
- When s arrives, return the joined tuple rs.

By instead using the pair of STAIRs $R.a$ and $S.a$, this would be done as follows:

- Insert r into $R.a$ with the call $insert(R.a, r)$ and probe for $r.a$ in $S.a$ with the call $probe(S.a, r.a)$.
- Insert s into $S.a$ with the call $insert(S.a, s)$ and probe for $s.a$ in $R.a$ with the call $probe(R.a, s.a)$.

The first call to *probe* will return nothing since there are no inserted tuples in $S.a$ at that time. The second call to *probe* will return the tuple r which can then be joined with s.

Now, in order to allow the eddy to do state manipulation, the STAIRs also support the following operations:

- $demotion(R.a, t, t')$ replaces t by t' in $R.a$, where t' is a sub-tuple of t.
- $promotion(R.a, t, S.b)$

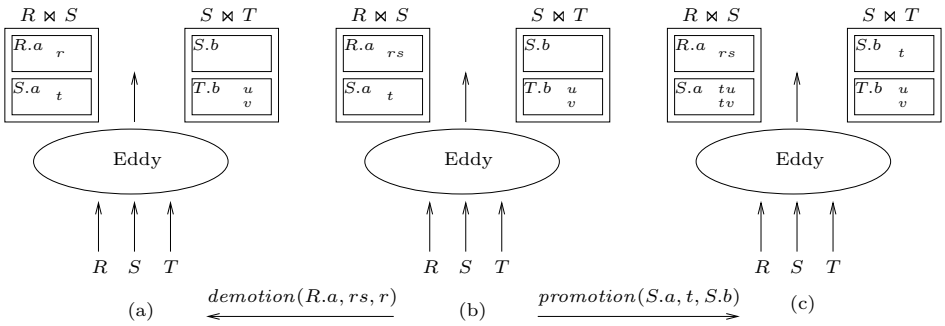

Figure 3: Three different states of an eddy with STAIRs. Starting from the state in (b), the state in (a) shows the eddy after the call $demotion(R.a, rs, r)$, and the state in (c) shows the eddy after the call $promotion(S.a, t, S.b)$.

- removes t from $R.a$;
- inserts t into $S.b$;
- probes $T.b$ with $t.b$, where $T.b$ is the other STAIR operator of a join $S \bowtie T$;
- inserts the resulting matches (which are super-tuples of t) back into $R.a$.

These two operations are only valid when certain preconditions are fulfilled. We will not spell out these preconditions here but they can be found in [2]. Intuitively, the call $demotion(R.a, t, t')$ undoes a join that resulted in the tuple t, where one of the joined tuples is t'. The call $promotion(R.a, t, S.b)$ reroutes the tuple t, initially routed with respect to the join ordering $(R \bowtie S) \bowtie T$, with respect to the new join ordering $R \bowtie (S \bowtie T)$.

In Figure 3 we see the eddy of Figure 2 where the two joins are replaced by two pairs of STAIR operators. Starting from the eddy's state in Figure 3(b), we see in Figure 3(a) the eddy after the call $demotion(R.a, rs, r)$. Indeed, the tuple rs in $R.a$ is replaced by the sub-tuple r. In Figure 3(c), we see the eddy after the call $promotion(S.a, t, S.b)$. Indeed, the tuple t has now been rerouted with respect to the join ordering $R \bowtie (S \bowtie T)$. We see this since, in Figure 3(c), the joined tuples tu and tv are inserted in $S.a$ and t is inserted in $S.b$.

By using *demotion* and *promotion* it is thus possible to manipulate the state of the eddy, and a state forcing the eddy to make bad routing decisions can be changed such that the eddy can (continue to) make good routing decisions.

4 Conclusion

In this overview we have presented part of the ideas in [1, 2] where the main goal is to provide more efficient query operators. This was realised by providing the eddy operator [1] which is an adaptive query operator that may choose different

query plans on a tuple-by-tuple basis. Since the eddy operator may suffer from state buffering, forcing it to make bad routing decisions, the extension to eddy named STAIR was introduced in [2]. With this extension, the operator is truly adaptive since it may manipulate its state, and thus overcome bad routing decisions.

References

[1] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 261–272, 2000.

[2] Amol Deshpande and Joseph M. Hellerstein. Lifting the burden of history from adaptive query processing. In *Proceedings of 30th International Conference on Very Large Data Bases*, pages 948–959, 2004.

[3] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using state modules for adaptive query processing. In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, 2003.