Chapter 3

Runtime Plan Optimization: Switching between Automaton and Algebra Processing Styles

In the previous chapter, we have illustrated that the decisions regarding which patterns to be retrieved in the automaton or out of the automaton can have significant impact on the performance of query evaluation. In this chapter, we explore how to get a good plan taking advantage of this optimization opportunity.

3.1 Solution Space

We provide a set of rewrite rules in Raindrop. From an initial plan, by repeatedly applying the rewrite rules, we can get a batch of alternative plans that compose the search space. We now describe these rewrite rules. In this chapter, we use the query shown in Figure 3.1 as the running example. Figure 3.2 shows a plan, which
retrieves all pattern in the automaton, for this query.

```xml
for $a in stream(open_auctions)/auctions/auction[reserve]
$b in $a/seller, $c in $a/bidder
Where $b//profile contains 'frequent' and $c//zipcode='01609'
return
<auction> {$b, $c} </auction>
```

Figure 3.1: Example Query for Automaton-in-or-out Optimization

### 3.1.1 Token-or-Node Mode Change Rules

The token-or-node mode change rules, as described in Section 2.4, change the modes (i.e., on tokens or on nodes) of pattern retrieval. This is the key rewrite rule for generating alternative plans in our solution space. Since a pattern retrieval on tokens (resp. on nodes) is performed in the automaton (resp. out of the automaton), we also say this rule pulls pattern retrieval out of the automaton or pushes patterns retrieval into the automaton. For ease of reading, we recap these rules briefly.

Figures 3.3 and 3.4 show the token-or-node mode change rules in two circumstances. In Figure 3.3, no StructuralJoin$_{\text{col}1}$ exists in the top plan so that a StructuralJoin$_{\text{col}1}$ is introduced when $\text{col}2 = \text{col}1/\text{path}1$ is pushed down. In Figure 3.4, a StructuralJoin$_{\text{col}1}$ exists in the top plan so that no new StructuralJoin is introduced when $\text{col}3 = \text{col}1/\text{path}2$ is pushed down. Figure 3.5 further shows a rule that eliminates an unnecessary Extract$_{\text{col}0}\text{col}1$ operator when $\text{col}1$ is not consumed by any non-automaton operators.

An interesting feature of the mode change rules is that when we push a pattern retrieval, say $\text{col}2 = \text{col}1/\text{path}$, into the automaton, the resultant TokenNav$_{\text{col}1, \text{path}}\text{col}2$ can only be placed in one unique position, i.e., right on top of a TokenNav that re-
3.1. SOLUTION SPACE

tries $col1$ (e.g., in Figure 3.4, $TokenNav_{col1,path2}col3$ has to be placed above $TokenNav_{col0,path3}col1$). In other words, $TokenNav_{col1,path2,col2}$ cannot be commuted with any other operators. This is because of the on-the-fly access nature of stream processing. Tokens cannot be accessed twice, $TokenNav_{col1,path2,col2}$ must be immediately evaluated on the tokens that compose bindings of $col1$.

In contrast, when we pull $col2 = col1/path$ out of the automaton, the resultant $NodeNav_{col1,path2}col2$ may be placed in multiple positions. For example, Figure 3.6 shows a plan after we pull $TokenNav_{a,auction}col1$ out of the plan in Fig-

\footnote{Tokens can however be stored as XML element nodes which can be repeatedly accessed.}
3.1. SOLUTION SPACE

Figure 3.3: Mode Change with Introducing/Eliminating StructuralJoin

Figure 3.4: Mode Change without Introducing/Eliminating StructuralJoin
3.1. SOLUTION SPACE

Figure 3.5: Eliminate $Extract_{col0}col1$ when no Regular Tuple-based Operator Consumes $col1$
3.1. SOLUTION SPACE

If later we pull out $d = a/reserve$, the resultant $NavNest_{a, /reserve}$ is placed by default between $ExtractUnnest_{a}$ and $NavUnnest_{a, /seller}$.

However it can also be placed for example between $NavUnnest_{a, /seller}$ and $NavNest_{b, /profile}$, because $NavNest_{a, /seller}$ still outputs tuples carrying cells bound to $a$.

![Diagram](image)

Figure 3.6: Plan Derived from the Pull-out of $TokenNav_{a, /seller}$ from Plan in Figure 3.2

Operator commuting has been long studied as an important optimization opportunity [19, 45]. This motivates us to introduce a second kind of rewrite rules in the next section to explore this opportunity.

3.1.2 Operator Commuting Rules

We now list the commuting rules. We use $Op_c$ to represent a Select or a NodeNav operator. $c$ represents the selection predicate if $Op$ is a Select operator, or the
3.1. SOLUTION SPACE

path expression if $Op$ is a $NodeNav$ operator. $P$, $P_1$, and $P_2$ in the rewrite rules represent subplans. We also use $\triangleright S J$ to represent a $StructuralJoin$ operator.

**Commuting $Op_{c_1}$ with $Op_{c_2}$:**

$Op_{c_1}(Op_{c_2}(P)) = Op_{c_2}(Op_{c_1}(P))$ when both $c_1$ and $c_2$ involve only columns output generated by a subplan $P$.

**Commuting $Op_c$ with $StructuralJoin$:**

$Op_c(P_1 \triangleright S J P_2) = (Op_c(P_1)) \triangleright S J P_2$ when $c$ involves only columns output by $P_1$.

Figures 3.7, 3.8 and 3.9 show the examples of commuting a $NodeNav$ operator with a $Select$, another $NodeNav$ and a $StructuralJoin$ operator respectively. A $NodeNav_{\text{col1,path1}}$ can commute with any automaton-outside operator as long as the $Extract$ operator that extracts $\text{col1}$ is still placed under $NodeNav_{\text{col1,path1}}$ after the commuting.

### 3.1.3 Input Subplan Reordering Rule

After we have determined where to place a $NodeNav$ operator, we can have further optimization decisions to make. For example, in Figure 3.6, according the execution style of $StructuralJoin$ operators as described in Section 2.5.4, when $StructuralJoin_{\text{Sc}}$ is invoked as a $<\text{/auction}>$ is encountered, only the three highlighted operators can have data in their output queues (note that the output of any descendant operator of $StructuralJoin_{\text{Sc}}$ must have all been consumed when $<\text{/bidder}>$ was encountered).

For each of the three operators, denoted as $op$, the intermediate operators be-
3.1. SOLUTION SPACE

Figure 3.7: Commuting $NodeNav_{\text{col2,path2}}$ with $Select_{\text{col1}}$

Figure 3.8: Commuting $NodeNav_{\text{col1,path1}}$ with $StructuralJoin$
3.1. SOLUTION SPACE

Figure 3.9: Commuting $\text{NodeNav}_{\text{col1, path1}} \text{col2}$ with $\text{NodeNav}_{\text{col3, path3}} \text{col4}$

tween $\text{op}$ and $\text{StructuralJoin}_{sa}$ must be evaluated when $\text{StructuralJoin}_{sa}$ is invoked. We call the intermediate operators between $\text{op}$ and $\text{StructuralJoin}_{sa}$ an input subplan of $\text{StructuralJoin}_{sa}$ and $\text{op}$ the entry operator of this input subplan.

For example, the three dashed boxes in Figure 3.6 contain three input subplans of $\text{StructuralJoin}_{sa}$ with entry operators $\text{ExtractNest}_{sa}d$, $\text{ExtractUnnest}_{sa}a$, and $\text{StructuralJoin}_{sc}$ respectively. Even though there is no intermediate operator between $\text{ExtractNest}_{sa}d$ and $\text{StructuralJoin}_{sa}$, for uniformity, we say $\text{Extractnest}_{sa}d$ is the entry operator of an empty input subplan of $\text{StructuralJoin}_{sa}$.

The method $\text{ancestorUpstreamDriven}$ in Algorithm 3 in Section 2.6 describes the process of evaluating these input subplans. When an $<\text{auction}>$ is encountered, $\text{StructuralJoin}_{sa}$ is invoked. It then in turn invokes its input operators (lines 7 - 9 in Algorithm 3). Each such input operator again invokes its input op-
erator. Finally, the entry operator is invoked by its parent operator. Therefore the data in the output queue of the entry operator are consumed all the way through the input subplan. In this way, an input subplan is thoroughly evaluated. After all three input subplans have been evaluated, $\text{StructuralJoin}^a$ performs Cartesian products on the output of these input subplans (line 10 in Algorithm 3).

We now propose to further optimize to this process. Algorithm 5 improves Algorithm 3 in two ways.

**Precheck of Output of Entry Operators.** The first improvement is that when $\text{StructuralJoin}^a$ is invoked, it checks whether all entry operators have generated some output during the processing of the current binding of $\$a$ (lines 7 - 12 in Algorithm 5). Only if yes, $\text{StructuralJoin}^a$ goes on to evaluate the input subplans. For example, suppose $\text{ExtractNest}^a_d$ does not have output when checked, i.e., the current auction element does not have a reserve child element, then we can save the evaluation of the input subplans contained in the two dashed boxes.

**Immediate Stop at Empty Output of Input Subplans.** The second improvement is that when we evaluate the input subplans one by one, if a subplan does not generate output, we immediately stop evaluating the rest subplans (lines 17 - 19) since it is guaranteed that the $\text{StructuralJoin}$ would not output anything. We however need to assure that all unconsumed data are cleaned up. First, for those input subplans that have already generated output before we stop the evaluation, we clean up their output (lines 28 - 30 in Algorithm 5). Second, for those input subplans that have not been evaluated yet, we clean up their input, i.e., the data generated by their entry operators (lines 31 - 33 in Algorithm 5). This assures correctness as no old data will be mixed with the new data that will be generated.
3.1. SOLUTION SPACE

Algorithm 5 Optimized In-Time Structural Join (Compared to Algorithm 3)

```java
public class StructuralJoin {
    public ancestorUpstreamDriven()
    {
        boolean allEntryHaveResults = TRUE;
        boolean allSubplanHaveResults = TRUE;
        int i;
        List inputTuples[];
        int n = number of input operators of this StructuralJoin;
        //Precheck of Output of Entry Operators
        for each entry operator entryOp of input subplans do
            if entryOp has no data in its output queue then
                allEntryHaveResults = FALSE;
                break;
            end if
        end for
        if allEntryHaveResults then
            for (i = 1; i ≤ n; i++) do
                Let inputOp denotes the i'^th' input operator;
                List curInputTuples = output generated when inputOp is evaluated;
                //Immediate Stop at Empty Output of Input Subplans
                if curInputTuples are empty then
                    allSubplanHaveResults = FALSE;
                    break;
                else
                    inputTuples[i] = curInputTuples;
                end if
            end for
        end if
        if allSubplanHaveResults then
            outputTuples = join inputTuples[1], inputTuples[1], ..., and inputTuples[n];
        else
            for (int j = 1; j ≤ i; j++) do
                clean up output queue of the j'^th' input operator.
            end for
            for (int j = i + 1; j ≤ n; j++) do
                clean up output queue of the entry operator of the j'^th' input subplan.
            end for
        end if
        ... //lines 13 - 21 in Algorithm 3 in Section 2.6 in Chapter 2
    }
}
```
The order in which we evaluate the input subplans is important for the efficiency. For example, in Figure 3.6, suppose a seller seldom has a profile, then the second input plan should be evaluated before the third input plan. Therefore if we find that the second input plan does not generate any output within a binding of $a$, we do not need to evaluate the third input plan. This can lead to significant cost savings when there is a large number of bidder elements in an auction. We therefore offer a third rewrite rule called input subplan reordering. This rule switches the order of the input subplans whose topmost operators are $op_1$ and $op_2$ respectively.

**Reordering Input Plans:**

\[ op_1 \bowtie_{S,J} op_2 = op_2 \bowtie_{S,J} op_1. \]

This rule is graphically shown in Figure 3.10. In Figure 3.10, we assume the input subplans are evaluated from left to right. We change the order of the input subplans in the top plan and get the bottom plan.
3.2. COST MODEL

3.1.4 Relationships among Rewrite Rules

The operator-commuting and input-subplan-reordering rules are designed to complement the token-or-node mode change rules. The comparison of the performance when a pattern is retrieved in or out of the automaton should be fair. That means both the automaton processing and non-automaton processing should be optimized. Given a set of patterns to be retrieved in the automaton, the automaton part of the plan is uniquely determined. There are however alternatives for the non-automaton part of the plan. The operator-commuting and input-subplan-reordering rules are then applied to optimize the non-automaton part of the plan.

3.2 Cost Model

In order to be able to compare two alternative Raindrop plans, we now propose a cost model. In traditional databases, the cost of a plan is defined as the processing time on the whole input data. Since the input stream can possibly be infinite, we need to define the cost of the plan as the processing time on a finite input unit. Because we never allow pulling out the bottommost TokenNav operator in order not to buffer the complete incoming stream (refer to Section 2.4.1), all alternatives have the same bottommost TokenNav operator. We therefore define the cost of a plan (resp. an operator) as the average processing time on processing the data that originate from one destination element located by the bottommost TokenNav operator. For example, the cost of the plan in Figure 3.2 is the average time spent on processing one auction element; the cost of $TokenNav_{$b,//profile}$e is the average
3.2. COST MODEL

...time for locating all profile elements within one auction element. For simplicity, in the rest of this chapter, we refer to the destination element located by the bottommost TokenNav as a bottom input element.

We propose our cost model for a scenario with the following features: (1) the statistics are unavailable before the stream comes in; and (2) the query however is known beforehand, i.e., users preregister their queries before the stream arrives. In this scenario, we can run an initial plan of the query on the incoming stream and collect the statistics needed for this particular query. We will further discuss for other scenarios, which parts in our proposed cost model fit and which parts need to be extended in Section 3.2.5.

As described in Section 2.2, besides XML specific operators such as navigation, Raindrop also supports SQL-like operators such as Select, Join, Groupby, Orderby, Union, Difference and Intersect. In the first step of cost-based optimization for Raindrop plans, we consider only Select operator among the SQL-like operators. We can however extend to support the other SQL-like operators in future work. Note that the cost model for the SQL-like operators is not a major challenge since it has been widely studied in relational databases. The novel aspect of Raindrop cost model lies more in costing the automaton, which is little studied before.

3.2.1 Unit Costs of Automaton-Outside Operators

In Raindrop implementation, the cost of a unary automaton-outside operator is linear in the number of its input tuples. Also, the cost of the multi-way oper-
3.2. COST MODEL

ator, i.e., StructuralJoin, is linear in the product of the number of input tuples from each of its child operators. In other words, in current Raindrop implementation, given an automaton-outside operator op that has n child operators childOp1, childOp2, ..., childOpn, its cost can be expressed as $|childOp_1| \times |childOp_2| \times \ldots \times |childOp_n| \times UnitCost(op)$ where $|childOp_i|$ (1 < i < n) denotes the cardinality of the input originated from childOpi during the processing of a bottom input element; and UnitCost(op) is the processing time on each input tuple.

We further assume that the unit cost of an operator is not affected by how many number of input tuples the operator processes each time. Aurora [20] observes “intra-operator non-linearity” of tuple processing by an operator. That is, the unit cost of tuple processing may decrease as the number of tuples for processing increases. According to [20], this reduction in unit cost may arise due to two reasons. First, an operator may optimize its execution better with larger number of tuples available for processing. For example, merge joins can be used instead of nested loop joins for larger number of input tuples. Second, the total number of calls to the operator code decreases, cutting down the overhead of function calling. In Raindrop plans, operators do not have different evaluation strategies to cater to larger number or smaller number of tuples. Therefore, “intra-operator non-linearity” cannot arise because of the first reason mentioned above. Most cost models, relational [63?] or XML [6, 57], ignore such “non-linearity” arising because of the second reason. This is because it is hard to quantify the overhead of operator code which is very low level. We assume the same in Raindrop.

An important question to ask is, given an operator op, is it possible to observe its UnitCost(op) during the execution of an arbitrary plan? If yes, we can directly use this UnitCost(op) observed during the execution of an initial plan. If not, we
then have to analyze what factors contribute to \( \text{UnitCost}(op) \), i.e., cost models for such operators have to be defined at a lower granularity than \( \text{UnitCost}(op) \). For different operators, the answer is analyzed below:

1). A Select operator, when appearing in one plan, must appear in all other equivalent alternative plans because we do not provide any rewrite rule to eliminate a Select operator. Therefore, no matter what the currently running plan is, \( \text{UnitCost} \) of a Select operator is always observable. Since we assume that the \( \text{UnitCost}(op) \) is not affected by how many number of input tuples are processed each time the operator code is called, \( \text{UnitCost}(op) \) observed in a currently running plan is the same as that in any other plans.

2). A NodeNav operator does not appear in all plans due to the token-or-node mode change rule. Also, a StructuralJoin operator may not necessarily appear in every plan, e.g., \( \text{StructuralJoin}_{\text{col1}} \) appears in the bottom plan but not the top plan in Figure 2.6 in Section 2.4.2 in Chapter 2. Therefore, \( \text{UnitCost} \)'s of these two operators are not always observable in a currently running plan.

In summary, we may have to estimate the unit cost of a NodeNav or a StructuralJoin for costing a plan other than the currently running plan but this is not necessary for a Select operator. Therefore in the rest of this section, we analyze how to estimate the \( \text{UnitCost} \) for the NodeNav and StructuralJoin operators only. Table 3.1 gives the notations used for estimating these \( \text{UnitCost} \).

**UnitCost of NodeNav.** \( \text{UnitCost}(\text{NodeNav}_{u,p,v}) \) is the time NodeNav spends on navigating into the tree rooted at a node which is a binding of \( u \) to find all
3.2. COST MODEL

<table>
<thead>
<tr>
<th>Notation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_p[i]$</td>
<td>for $NodeNav_{u,p}v$, we use $p[i]$ to denote the $i^{th}$ navigation step on path $p$. $p[0]$ denotes the binding of $u$. $n_p[i]$ denotes average number of children of a binding of $p[i]$ within a binding of $u$</td>
</tr>
<tr>
<td>$w_p[i]$</td>
<td>for $NodeNav_{u,p}v$, $w_p[i]$ denotes average number of a binding of $p[i]$ within a binding of $u$</td>
</tr>
<tr>
<td>$C_{visit}$</td>
<td>time for visiting one node in an XML element tree</td>
</tr>
<tr>
<td>$C_{bicartesian}$</td>
<td>cost of performing a binary cartesian product, one input tuple from either side</td>
</tr>
</tbody>
</table>

Table 3.1: Notations Used in Defining $UnitCost$'s for $NodeNav$ and $StructureJoin$

the nodes that are bindings of $v$. Suppose $p = p[1]/p[2]/.../p[n]$ where $p[i]$ $(1 \leq i \leq n)$ is either a navigation step or a descendant axis “//” (for uniformity, we also view “//” as a special navigation step). To match the $i^{th}$ navigation step, every child of bindings of the $i - 1^{th}$ navigation step is visited. The number of these child nodes within a binding of $u$ is $n_{p[i-1]}w_{p[i-1]}$. Thus the time spent on finding $p[i]$ is $n_{p[i-1]}w_{p[i-1]}C_{visit}$. We then have the below equation.

Equation 1 $UnitCost(NodeNav_{u,p[1]/p[2]/.../p[n]}v) = \sum_{i=1}^{n} n_{p[i-1]}w_{p[i-1]}C_{visit}$.

$UnitCost$ of $StructureJoin$. Suppose a $StructureJoin$ has $n$ child operators $childOp_1$, $childOp_2$, ..., $childOp_n$. The $UnitCost$ of $StructureJoin$ is defined as the time spent on cartesian producting a tuple output by $childOp_1$, a tuple output by $childOp_2$, ..., with a tuple output by $childOp_n$. This time spent on the cartesian product may differ when $n$ differs. The values of $n$ for a $StructureJoin_v$ operator in different alternative plans can be different. $n$ can increase after the mode change of a $NodeNav$ operator (see Figure 2.6 in Section 3.1.1). We ignore this difference to avoid an overcomplicated cost model. We therefore use the unit cost of performing a binary Cartesian product (i.e., $n = 2$) as the general unit cost of a $StructureJoin$. We then have the below equation.
3.2. COST MODEL

Equation 2  \( \text{UnitCost}(\text{StructuralJoin}) = C_{\text{bicartesian}} \).

3.2.2 Costs of Input Subplans of StructuralJoin

We have studied how to get \( \text{UnitCost}(op) \) for an automaton-outside operator \( op \). Now we consider how to compute the cost of \( op \), denoted as \( \text{Cost}(op) \). As mentioned in Section 3.2.1, \( \text{Cost}(op) = |\text{childOp}_1| \times |\text{childOp}_2| \times \ldots \times |\text{childOp}_n| \times \text{UnitCost}(op) \). \(|\text{childOp}_1| \times |\text{childOp}_2| \times \ldots \times |\text{childOp}_n| \) is the amount of input to \( op \) during the processing of a bottom input element. In a traditional plan, the amount of data that needs to be processed by an operator is only affected by how much data is filtered by its descendant operators (i.e., the selectivity of its descendant operators). However, when a \( \text{StructuralJoin} \) is invoked, an input subplan is executed only when its left sibling subplans have all generated some output. Therefore the amount of data that needs to be processed by an input subplan is also affected by the likelihood of the left sibling subplans having generated some output.

We now define two concepts, \( \text{selectivity} \) and \( \text{non-empty-output probability} \), of operators. We also define a third concept \( \text{entry plan} \) for entry operators. These concepts are used to compute the cost of an input subplan.

**Selectivity**: The selectivity of an operator \( op \), denoted as \( \sigma(op) \) is defined as below:

1. If \( op \) is a \( \text{TokenNav}_u,p$u,v \) or \( \text{Extract}_u$u,v \), \( \sigma(op) \) is the average number of bindings of \$v \) generated within a binding of \$u \).

2. If \( op \) is a \( \text{Select}, \text{NodeNav} \) or \( \text{StructuralJoin} \), \( \sigma(op) \) is defined as in the traditional databases. Suppose \( op \) has \( n \) child operators, \( \sigma(op) \) is defined as

\[
\prod_{i=1}^{n} \frac{\text{cardinality of output}}{\text{cardinality of input from } v^{th} \text{ child operator of } op}.
\]
Non-empty-result Probability: The non-empty-result probability of an operator \( op \) is denoted as \( P_{\not\Rightarrow \emptyset}(op) \). “\( \not\Rightarrow \emptyset \)” in the notation means “not generating an empty result”, i.e., generating some result. It is defined as below:

1. If \( op \) is a \( \text{TokenNav}_{u,p} \), \( P_{\not\Rightarrow \emptyset}(op) \) is the probability of a binding of \( u \) containing at least one binding of \( v \).

2. If \( op \) is a \( \text{Select} \) or \( \text{NodeNav} \), \( P_{\not\Rightarrow \emptyset}(op) \) is the probability of \( op \) generating some output during the processing of one input tuple.

Entry Plan: As described in Section 3.1.3, a \( \text{StructuralJoin}_{v} \) has several entry operators. For example, in Figure 3.6, the three highlighted operators are the entry operators of \( \text{StructuralJoin}_{a} \). There are intermediate operators between an entry operator and the \( \text{TokenNav} \) operator that retrieves \( v \). We call the plan consisting of these intermediate operators (including the entry operator) an entry plan. In Figure 3.6, there are five intermediate operators between the entry operator \( \text{StructuralJoin}_{c} \) and \( \text{TokenNav}_{a,auctions/auction} a \), i.e., \( \text{StructuralJoin}_{c} \), \( \text{ExtractUnnest}_{a,c} \), \( \text{ExtractNest}_{c,f} \), \( \text{TokenNav}_{c,zipcode} f \), and \( \text{TokenNav}_{a,bidder} c \). We say the plan composed of these five operators an entry plan of the entry operator \( \text{StructuralJoin}_{c} \). We use the function \( \text{entryPlan}(op) \) to denote the entry plan of an entry operator \( op \).

Assume the input subplans of \( \text{StructuralJoin}_{v} \) from left to right are \( \text{subplan}_1 \), \( \text{subplan}_2 \), ..., \( \text{subplan}_n \) with entry operators \( \text{entry}_1 \), \( \text{entry}_2 \), ..., \( \text{entry}_n \) respectively. Equation 3 computes the cost of \( \text{subplan}_i \) (\( 1 \leq i \leq n \)).

Equation 3 \( \text{Cost}(\text{subplan}_i \text{ of } \text{StructuralJoin}_{v}) \)

\[
\text{number of bindings of } v \text{ within one bottom input element (1)}
\]

\[\text{number of bindings of } v \text{ within one bottom input element (1)}\]
3.2. COST MODEL

\[ \times \text{evaluation time of subplan}_i \text{ on input generated within a binding of } \$v \ (2) \]

\[ = \prod_{op \in \text{operator set between bottommost TokenNav and TokenNav that retrieves } \$v} \sigma(op) \ (3) \]

\[ \times \text{probability of subplan}_i \text{ being evaluated } (4) \]

\[ \times \text{amount of input tuples to subplan}_i \text{ within a binding of } \$v \ (5) \]

\[ \times \text{evaluation time of subplan}_i \text{ on one input tuple } (6) \]

\[ = \prod_{op \in \text{operator set between bottommost TokenNav and TokenNav that retrieves } \$v} \sigma(op) \ (7) \]

\[ \times P_{\neq \emptyset}(\text{entry}_1)P_{\neq \emptyset}(\text{entry}_2)...P_{\neq \emptyset}(\text{entry}_n) \ (8.a) \]

\[ \times P_{\neq \emptyset}(\text{subplan}_1)...P_{\neq \emptyset}(\text{subplan}_{i-1}) \ (8.b) \]

\[ \times \sigma(\text{entryPlan}(\text{entry}_i)) \ (9) \]

\[ \times \text{UnitCost}(\text{subplan}_i) \ (10) \]

In Equation 3, Expression (1) is expanded into Expression (3). When we say “operator set between bottommost TokenNav and the TokenNav that retrieves \$v”, the set does not include bottommost TokenNav but it includes the TokenNav that retrieves \$v. Any operator in the set is a TokenNav that retrieves an ancestor pattern of \$v or \$v itself. For example, suppose we want to cost an input subplan of \textit{StructuralJoin}\_$_c$ in Figure 3.6. To compute the number of bindings of \$c$ in the bottom input element, the operators set between the bottommost TokenNav (i.e., TokenNav$_{a,\text{auctions/auction}}\$a$) and the TokenNav that retrieves \$b$ (i.e., TokenNav$_{a,\text{seller}c}$) is \{TokenNav$_{a,\text{seller}c}$\}. Expression (3) is then expanded as \(\sigma(\text{TokenNav}_{a,\text{seller}c})\), i.e., the number of bindings of \$c$ within a binding of \$a$.

Expression (2) is expanded into Expressions (4) (5) and (6). Expression (4)
3.2. COST MODEL

later is expanded into Expressions (8.a) and (8.b). Expression (8.a) gives the probability of all entry operators generating output while Expression (8.b) gives the possibility of all left sibling input plans of subplan \(_i\) generating output.

Finally, Expressions (5) and (6) are expanded into Expressions (9) and (10) respectively. The average number of tuples generated by entry \(_i\) within a binding of \(v\) is the selectivity of the entry plan of entry \(_i\), i.e., \(\sigma(entryPlan(entry_i))\) in Expression (9). The unit cost of processing one input tuple of subplan \(_i\) is UnitCost(subplan \(_i\)) in Expression (10).

\(\sigma(entryPlan(entry_i))\) and UnitCost(subplan \(_i\)) require us to compute the selectivity and the cost of a plan respectively. This can be computed exactly as in traditional databases. We compute the selectivity of a plan as below.

1). For a plan \(P = P_A(P_B)\) which means subplan \(P_A\) consumes output of subplan \(P_B\), \(\sigma(P) = \sigma(P_B) \times \sigma(P_A)\); and \(Cost(P) = n \times UnitCost(P_B) + n \times \sigma(P_B) \times UnitCost(P_A)\) where \(n\) is the number of input to \(P_B\).

2). For a plan \(P = P_A JoinOp P_B\) which means subplan \(P_A\) is joined with subplan \(P_B\) by JoinOp, \(\sigma(P) = \sigma(P_A) \times \sigma(P_B) \times \sigma(JoinOp)\); and \(Cost(P) = n_A \times UnitCost(P_A) + n_B \times UnitCost(P_B) + n_A \times n_B \times \sigma(JoinOp) \times UnitCost(JoinOp)\) where \(n_A\) and \(n_B\) are the number of input tuples to \(P_A\) and \(P_B\) respectively.

By breaking a bigger plan into smaller subplans, we can eventually compute the selectivity/cost of a plan from the selectivity/cost of its operators.
3.2.3 Costs of Automaton-Inside Operators

In the previous section we have discussed how to compute costs for automaton-outside operators. We now describe how to compute the costs for the automaton-inside operators. We first briefly recap how an automaton is used to retrieve patterns while more details can be found in Section 2.5. An automaton behaves as below:

1). When an incoming token is a start tag:
   a. If the stack top is not empty, the incoming token is looked up in the transition entries of every state at the stack top. The automaton pushes the states that are transitioned to onto the stack. If no states are transitioned to, the automaton pushes an empty set (denoted as $\emptyset$) onto the stack.
   b. When the stack top contains an empty set, the automaton directly pushes another empty set onto the stack without any lookup.

2). When an incoming token is an end tag: the automaton pops the states at the stack top off the stack.

3). When an incoming token is a PCDATA token, the automaton makes no change to the stack.

4). An incoming token (start tag, end tag or PCDATA token) is stored if required by an $Extract$ operator.

When costing a pattern retrieval, we need to be careful with “amortized” computations. For example, in Figure 3.11, when a stack top contains instances of
Figure 3.11: Automaton of Plan in Figure 3.2 and Stack Snapshots

$q_4$ and $q_5$ (see the rightmost stack), an incoming $<$seller$>$ will lead to a stack backtrack. However we cannot solely assign this backtracking cost to the pattern retrieval $a/seller$. This is for two reasons. First, even if the query does not ask for $a/seller$, backtracking is still needed when $<$seller$>$ is encountered in order to restore the stack to the status before the matching $<$seller$>$ has been encountered. Second, the backtracking cost is a constant, i.e., it is not affected by which states are popped or the number of states popped. For example, in Java implementation, we can simply move the reference to the stack top one level down to accomplish the state pop-off.

To avoid repeatedly costing the same amortized computations, we analyze the cost of retrieving a pattern $p$ by comparing the cost of running a stream on an automaton $A_{with}$ and the cost of running the same stream on another automaton $A_{without}$. $A_{with}$ denotes an automaton that encodes $v/p$ and all the ancestor patterns of $v/p$ (e.g., the ancestor patterns of $b/profile$ are $a = auctions/auction$ and $b = a/bidder$). $A_{without}$ encodes only the ancestor patterns of $v/p$. Since
3.2. COST MODEL

<table>
<thead>
<tr>
<th>Notation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q(A)$</td>
<td>states in an automaton $A$</td>
</tr>
<tr>
<td>$Q_{extract}(A)$</td>
<td>states associated with extraction operators, e.g., states $q_4$ and $q_7$ in Figure 3.11</td>
</tr>
<tr>
<td>$C_{nonEmp}$</td>
<td>cost of processing a start token when stack top is not empty</td>
</tr>
<tr>
<td>$C_{emp}$</td>
<td>cost of processing a start token when stack top is empty</td>
</tr>
<tr>
<td>$C_{backtrack}$</td>
<td>cost of popping off states at the stack top</td>
</tr>
<tr>
<td>$C_{extract}(q)$</td>
<td>cost of buffering elements, whose start tags activates state $q$, in a bottom input element</td>
</tr>
<tr>
<td>$n_{active}(q)$</td>
<td>the number of times that stack top contains a state $q$ when a start tag arrives in a bottom input element. Each such tag is the start tag of a child of an element that activates $q$</td>
</tr>
<tr>
<td>$n_{start}$, $n_{end}$</td>
<td>number of start or end tags in a bottom input element. $n(start) = n(end)$.</td>
</tr>
</tbody>
</table>

Table 3.2: Notations Used in Cost of Automaton-Inside Operators

$A_{with}$ and $A_{without}$ only differs in that $A_{with}$ retrieves an additional pattern $p$, the cost difference of running a stream on $A_{with}$ and $A_{without}$ is then the cost of retrieving $p$ in the stream.

We first study how to compute the cost of running a stream on an automaton. Given an automaton $A$ with a start state which is activated by a start tag of the bottom input element (i.e., $q_2$ in Figure 3.11), the cost of running a stream on the automaton $A$ is:

**Equation 4** $\text{Cost}(A) =$

\[
\begin{align*}
\text{state transiting cost for processing start tags} & \quad (1) \\
+ \text{stack backtracking cost for processing end tags} & \quad (2) \\
+ \text{extracting cost for processing tokens} & \quad (3)
\end{align*}
\]

Using the notations in Table 3.2, we can refine Equation 4 to Equation 5.

**Equation 5** $\text{Cost}(A) =$

\[
\begin{align*}
\sum_{q \in Q(A)} n_{active}(q) C_{nonEmp} & \quad (1.a) \\
+ [n_{start} \cdot \sum_{q \in Q(A)} n_{active}(q)] C_{emp} & \quad (1.b)
\end{align*}
\]
3.2. COST MODEL

\[ + n_{end} C_{backtrack} \tag{2} \]
\[ + \sum_{q \in Q_{extract(A)}} n_{active}(q) C_{extract}(q) \tag{3} \]
\[ = \sum_{q \in Q(A)} n_{active}(q) (C_{nonEmp} - C_{emp}) + n_{start}(C_{emp} + C_{backTrack}) \tag{4} \]
\[ + \sum_{q \in Q_{extract(A)}} n_{active}(q) C_{extract}(q) \tag{5} \]

Expression (1) in Equation 4 is expanded into Expressions (1.a) and (1.b) in Equation 5. A start tag activates more than one state only when “//” occurs in the query, namely, there are \( \lambda \) transitions and self transitions. For example, in Figure 3.11, if \( q_4 \) is at the stack top, \( q_5 \) must be at the stack top as well. Since \( \lambda \) transitions and self transitions usually is only a small portion of the transitions in an automaton, \( \sum_{q \in Q(A)} n_{active}(q) \) is approximately equal to the number of start tokens that are processed with a non-empty stack top. Therefore Expression (1.a) is the cost of processing start tags with a non-empty stack top.

The number of start tags that are processed with an empty stack top is \((n_{start} - \text{number of start tags that are processed with a non-empty stack top}) = (n_{start} - \sum_{q \in Q(A)} n_{active}(q))\). Expression (1.b) of Equation 5 thus is the cost of processing start tags with an empty stack top.

The cost of processing an end tag is equal to the cost of popping out the states at the stack top, namely, \( C_{backtrack} \). Since there are \( n_{end} \) end tags in a bottom input element, Expressions (2) in Equation 5 is the cost of processing end tags.

In Expression (3) in Equation 5, \( q \in Q_{extract(A)} \) is a state associated with an Extract operator. \( n_{active}(q) C_{extract}(q) \) then denotes the cost of storing the elements whose start tags activate \( q \). Therefore Expression (3) in Equation 5 is the total extraction cost.
3.2. COST MODEL

We now know how to compute the cost of running a stream on a given automaton. We can then compute the cost of a $TokenNav_{u,p}^v$ operator by computing $Cost(A_{\text{with}}) - Cost(A_{\text{without}})$, as shown in Equation 6. $A_p$ in the equation denotes the sub-automaton that encodes $v/p$ only.

**Equation 6** $Cost(TokenNav_{u,p}^v)$

\[
= Cost(A_{\text{with}}) - Cost(A_{\text{without}})
\]

\[
= \sum_{q \in Q(A_{\text{with}}) - Q(A_{\text{without}})} n_{\text{active}}(q) (Cost_{\text{nonEmp}} - Cost_{\text{emp}})
\]

\[
+ \sum_{q \in Q_{\text{extract}}(A_{\text{with}}) - Q_{\text{extract}}(A_{\text{without}})} n_{\text{active}}(q) Cost_{\text{extract}}(q)
\]

\[
= \sum_{q \in Q(A_p)} n_{\text{active}}(q) (Cost_{\text{nonEmp}} - Cost_{\text{emp}})
\]

\[
+ \sum_{q \in Q_{\text{extract}}(A_p)} n_{\text{active}}(q) Cost_{\text{extract}}(q)
\]

3.2.4 Cost Model Summary

The cost of a plan consists of two parts. The first part is the cost of all pattern retrieval performed in the automaton. We use Equation 6 to compute the cost of each pattern retrieval. The second part is the cost of automaton-outside operators. The automaton-outside operators can be divided into several disjunct groups, each group composed of a $StructuralJoin$ and its input subplans. We can use Equation 3 to compute the cost of each such group.

3.2.5 Discussion on Extension of Cost Models

In the beginning of Section 3.2, we mentioned that we assume the user query is known beforehand. With this assumption, we can then run an initial plan of the query and collect the statistics needed for this particular query. The impact of this query specific statistics collection mechanism is that for those operators that appear
in all alternative plans, we do not need to further analyze what factors contribute to their UnitCost because their UnitCost can be directly observed in the currently running plan.

There are two scenarios in which the above statistics collection mechanism does not fit. The first scenario is that the stream query engine has to process a large number of queries so that it cannot afford to collect specific statistics for each query. Statistics summary techniques [2, 84] are needed to achieve good scalability. The second scenario is that the user adds a new query after the stream starts to arrive. Of course we can still run an initial plan of this new query and collect statistics for it if scalability is not a concern here. Another solution is that we always summarize the statistics as the stream runs so that once a new query is added, we can immediately provide cost estimates and choose a plan for this query. This solution is essentially static optimization, i.e., getting the statistics, choosing a plan and running the chosen plan.

In summary, in both scenarios, we can estimate the cost of a plan from general statistics instead of specific statistics for this particular plan. A summary statistics collection mechanism may not observe the UnitCost of all Select operators in the plans. For example, operators that involve a “contain” function such as $Select_{se \text{ contains } “frequent”}$ are quite common in queries on text-centered XML document [7]. The query-specific statistics collection mechanism ensures that we can directly observe the UnitCost of such operators. In the summary statistics collection mechanism however, we need to enhance our cost model by analyzing what factors contribute to evaluating, for example, a “contain” function. Except such enhancement on analyzing the UnitCost's of the functions used in Select operators, all the other parts of our cost model still fit in the summary statistics collection
3.3 Combining Heuristics and Costs for Operator Commuting

The operator-commuting and input-subplan-reordering rules optimize the non-automaton part of a plan. The operator-commuting rule reorders two operators that have a parent-child relationship while the input-subplan-reordering rule reorders subplans that have a sibling relationship. We sometimes refer to these two rules as parent-child operator reordering and sibling operator reordering respectively. These two rules are not independent of each other. That means, optimizing a plan using one rule first and then optimizing the plan using the second rule does not ensure the resultant plan is overall optimal. We now give an example to illustrate the dependency relationship between the two rules.

Example 12 Without the input-subplan-reordering rule, the likelihood of an operator being executed is only decided by the selectivity of its descendant operators. For example, in Figure 3.2, if we push down \( Select_{f=01609} \) under \( Structural\ Join_{Sc} \), \( Select_{f=01609} \) will be placed between \( Extract\ Nest_{Sc} f \) and \( Structural\ Join_{Sc} \). \( Select_{f=01609} \) was above \( Extract\ Unnest_{Sa} c \) before the push-down. \( Extract\ Unnest_{Sa} c \) simply wraps tokens that are components of bindings of \( c \) into tuples. In other words, \( Extract\ Unnest_{Sa} c \) does not filter the input so that whether it is executed before \( Select_{f=01609} \) or in parallel with \( Select_{f=01609} \) does not affect the cost of \( Select_{f=01609} \). Therefore, pushing down \( Select_{f=01609} \) under \( Structural\ Join_{Sc} \) does not change the cost of \( Select_{f=01609} \). However
pushing down Select$_{f=01609}$ under StructuralJoin$_{sc}$ can decrease the cost of StructuralJoin$_{sc}$ since Select$_{f=01609}$ can filter input to StructuralJoin$_{sc}$.

In summary, Select$_{f=01609}$ would be pushed down under StructuralJoin$_{sc}$ when only the operator-commuting rule is considered. However if we in addition consider the input-subplan-reordering rule, leaving Select$_{f=01609}$ above the StructuralJoin$_{sc}$ operator as in Figure 3.2 may be better than pushing it down because we may be able to save the evaluation of Select$_{f=01609}$ when its left sibling subplans, for example, Select$_{se}$ contains "frequent", are very selective. Therefore, the parent-child operator relationships in a plan optimized without input-subplan-reordering rule are not necessarily the same as those in a plan optimized with the input-subplan-reordering rule.

Since the search space generated by only the token-or-node mode change rules can be already very large (a query with $n$ patterns can have up to $2^n$ alternative plans), we prefer to optimize the non-automaton part of a plan in a short time. We therefore use a search strategy that basically considers the operator-commuting and input-subplan-reordering rules independently, i.e., optimize in two phases. In the first phase, we optimize using only the operator-commuting rule on the initial plan and get a new plan. In the second phase, we then optimize the plan derived in the first phase using the input-subplan-reordering rule only. Such a strategy prevents a search space explosion compared to considering all combinations of applying both types of rules. It however is not exactly an independent search, since some operator-commuting decisions we make in the first phase target leaving opportunities for the later input-subplan-reordering optimization. For example, in Example 12, we may choose to place Select$_{f=01609}$ above StructuralJoin$_{sc}$. 
We present how to make the operator commuting decisions in this section while we present how to make the input-subplan-reordering decisions in Section 3.4.

### 3.3.1 Using both Heuristics and Costs for Operator Commuting

The operator commuting in Raindrop plans can be divided into two types. One is commuting Select-like operators with each other. Besides Select operators, NodeNav operators are special Select operators because a NodeNav$_{u,p}$ operator has only one child operator and filters out input tuples whose bindings of $u$ do not contain a path $p$. The second type is commuting Select-like operators with StructuralJoin operators. Note that in a Raindrop plan, StructuralJoin operators cannot be commuted with each other. Suppose we have StructuralJoin$_{u}$ and StructuralJoin$_{v}$ where $v$ is a descendant element within a $u$ (i.e., $v$ can be expressed as $v = u/p$). Because of the sequential manner of accessing token streams, a binding of $v$ must be completely accessed before the corresponding binding of $u$ has been completely accessed. That dictates that StructuralJoin$_{v}$ is always performed before StructuralJoin$_{u}$ on the data that are located within the same binding of $u$. That is to say, the order among ancestor and descendant StructuralJoin operators is fixed by the query semantics. Therefore StructuralJoin can only be commuted with Select and NodeNav operators.

For the first type of commuting, i.e., commuting Select-like operators with each other, we can utilize some existing techniques. [19] proposed a cost-based technique for determining the order of Select-like operators. The basic idea is to define a rank function on the operators. The operators are then evaluated in the ascending order of their rank functions. This order is guaranteed to be optimal. The rank function on a Select-like operator $op$ is defined as $rank(op) = \frac{\sigma(op) - 1}{UnitCost(op)}$ where...
3.3. COMBINING HEURISTICS AND COSTS FOR OPERATOR COMMUTATION

\( \sigma(op) \) is the selectivity of \( op \) (i.e., \( \frac{\text{number of output tuples}}{\text{number of input tuples}} \)) and \( \text{UnitCost}(op) \) is the cost of processing one input tuple in \( op \). Intuitively, this rank function indicates that if the operator has a low unit cost (i.e., processes one input tuple quickly) and a low selectivity (i.e., filters many of its input tuples), it should be executed early. Such a rank function based technique can also be used to commute \( \text{Select} \) or \( \text{NodeNav} \) operators in Raindrop.

For the second type of commuting, i.e., commuting \( \text{NodeNav} \) or \( \text{Select} \) with \( \text{StructuralJoin} \), the above cost-based technique no longer applies. [45] extends the rank function based technique in [19] to reorder \( \text{Select} \) and \( \text{Join} \) operators. [45] assumes certain properties of the \( \text{Select} \) and \( \text{Join} \) operators. Suppose a \( \text{Join} \) operator has two child operators \( \text{Sel}_1 \) and \( \text{Sel}_2 \). [45] assumes that commuting either \( \text{Select} \) operator with \( \text{Join} \) only affects the costs of these two operators commuted. For example, commuting \( \text{Sel}_1 \) with \( \text{Join} \) does not change the cost of \( \text{Sel}_2 \). Suppose a \( \text{StructuralJoin} \) also has two child \( \text{Select} \) operators \( \text{Sel}_1 \) and \( \text{Sel}_2 \) from left to right. Since we only evaluate \( \text{Sel}_2 \) after \( \text{Sel}_1 \) has generated output, the cost of \( \text{Sel}_2 \) is affected by the non-empty-probability of \( \text{Sel}_1 \). Commuting \( \text{Sel}_1 \) with \( \text{Join} \) can increase the cost of \( \text{Sel}_2 \). The assumption that only the costs of the operators involved in the commuting are changed is violated. Therefore, the rank function based technique does not work for commuting \( \text{Select} \) or \( \text{NodeNav} \) with \( \text{StructuralJoin} \). We instead propose heuristics for making decisions for such type of commuting.

In summary, we use the existing rank function based technique [19, 45] to commute \( \text{Select} \) or \( \text{NodeNav} \) operators in Raindrop while we propose heuristics to commute \( \text{NodeNav} \) or \( \text{Select} \) with \( \text{StructuralJoin} \). We have discussed how to compute \( \sigma(op) \) and \( \text{UnitCost}(op) \) in Section 3.2. We now describe our heuristics.
3.3. COMBINING HEURISTICS AND COSTS FOR OPERATOR COMMUTING

for commuting NodeNav or Select with StructuralJoin.

3.3.2 Heuristics for Commuting Select/NodeNav with StructuralJoin

We categorize StructuralJoin into three cases and propose heuristics for each case.

Case 1: StructuralJoin with Duplicate $v$ Output

**Heuristic 1:** Given a Select or NodeNav operator $op$ and a StructuralJoin, we place $op$ beneath the StructuralJoin if an ExtractUnnest $v$, $w$ or NavUnnest $v$, $p$, $w$ $(NavUnnest_{v,p} \neq op)$ exists in the plan.

For example, in Figure 3.6, NavUnnest $a$, seller $b$ would be placed beneath StructuralJoin $a$ since there exists a TokenNav $a$, bidder $c$ in the plan. Suppose we instead place NavUnnest $a$, seller $b$ above StructuralJoin $a$, if one auction has 10 bidder’s, StructuralJoin $a$ will output 10 tuples for this auction, one tuple for one different bidder. Then the same auction will be navigated into 10 times by NavUnnest $a$, seller $b$ to locate the seller.

Our experiment in Figure 2.27 in Section 2.7.2 in Chapter 2 has illustrated that such duplicate computations seriously degrade the plan performance. We therefore propose such a heuristic to avoid any duplicate computations.

Case 2: Intermediate StructuralJoin with Duplicate $v$ Output

**Heuristic 2:** We place a Select or NodeNav operator $op$ above a StructuralJoin $v$ if (1) except $op$, no ExtractUnnest $v$, $p$, $w$ nor NavUnnest $v$, $p$, $w$ exists in the plan and (2) other StructuralJoin operators exist above StructuralJoin $v$. 
This heuristic is designed to provide more opportunities for structural join related optimization. Two operators that belong to the input plans of different StructuralJoins have no impact on each other’s execution. For example, in Figure 3.2, suppose we place \( Select_{se \ contains \ “frequent”} \) and \( Select_{sf = “01609”} \) beneath \( StructuralJoin_{sb} \) and \( StructuralJoin_{sc} \) respectively. Each time when a \(<seller> \) is encountered and \( \text{profile} \) elements are located within this \( seller \) (resp. When a \(<bidder> \) is encountered and \( \text{zipcode} \) elements are located), \( Select_{se \ contains \ “frequent”} \) (resp. \( Select_{sf = “01609”} \) ) would have to be performed. Instead, consider the case in which we place \( Select_{se \ contains \ “frequent”} \) and \( Select_{sf = “01609”} \) above \( StructuralJoin_{sb} \) and \( StructuralJoin_{sc} \) as in Figure 3.2. When a \(<auction> \) is encountered, \( \text{bidder} \) elements would have to be found within this \( auction \) before \( Select_{se \ contains \ “frequent”} \) could possibly be performed. This is because \( StructuralJoin \) only evaluates its input subplans when all entry operators have generated output (see “precheck of output of entry operators” in Algorithm 3). Moreover, The evaluation of \( Select_{sf = “01609”} \) will not be performed if \( Select_{se \ contains \ “frequent”} \) does not generate any output (see ‘immediate stop at empty output of input subplans’ in Algorithm 3). Therefore, both \( Select_{se \ contains \ “frequent”} \) and \( Select_{sf = “01609”} \) are more likely to be avoided after the commuting.

Generally speaking, before this commuting rewriting, the \( Select \) and \( NodeNav \) operators were scattered in the input plans of different StructuralJoins. After the rewriting, these operators are “concentrated” into the subplans of less \( StructuralJoin \) operators. For example, all \( Select \) operators occur in the input subplans of \( StructuralJoin_{sa} \) in Figure 3.2. Such operators are then less likely to be evaluated because of the optimization techniques in \( StructuralJoin \).
3.3. COMBINING HEURISTICS AND COSTS FOR OPERATOR COMMUTING

Case 3: Topmost $\text{StructuralJoin}_{sv}$

**Heuristic 3:** We place a $\text{Select}$ or $\text{NodeNav}$ operator $op$ underneath a $\text{StructuralJoin}$ if no other $\text{StructuralJoin}$ operators exist above $\text{StructuralJoin}_{sv}$.

Placing $op$ above $\text{StructuralJoin}_{sv}$ does not open up any optimization opportunity for input subplan reordering as in the second case. It may even increase the cost of $\text{StructuralJoin}_{sv}$ because more data that could otherwise be filtered out by $\text{Select}$ or $\text{NodeNav}$ are now input to $\text{StructuralJoin}_{sv}$. Therefore for such a topmost $\text{StructuralJoin}$, $\text{Select}$ or $\text{NodeNav}$ operators should be placed underneath it. For example, in Figure 3.2, we keep both $\text{Select}_{se}$ contains “frequent” and $\text{Select}_{sf} = \text{01609}$ underneath $\text{StructuralJoin}_{sa}$, since it is the topmost $\text{StructuralJoin}$ in the plan.

3.3.3 Operator Commuting Algorithm

We now describe the algorithm that commutes the operators in a Raindrop plan. Lemma 1 shows an important property that is utilized in the algorithm.

**Lemma 1 Order of Applying Commuting Rules Being Insensitive:** Given two operators $op1$ and $op2$ ($op1$ or $op2$ can be either a $\text{Select}$ or a $\text{NodeNav}$), whether we commute $op1$ with $\text{StructuralJoin}$ or commute $op2$ with $\text{StructuralJoin}$ first does not affect the final outcome.

**Proof 1** We decide which category a $\text{StructuralJoin}$ belongs to by checking whether this $\text{StructuralJoin}$ can output duplicates and whether it is the topmost $\text{StructuralJoin}$. Commuting two operators would not eliminate an $\text{ExtractUnnest}_{sv}$,$w$ or $\text{NavUnnest}_{se,p}$,$w$. It therefore does not change the property of a $\text{StructuralJoin}_{sv}$, outputting dupli-
3.3. COMBINING HEURISTICS AND COSTS FOR OPERATOR COMMUTING

cates or not. Also, the commuting would not eliminate any StructuralJoin so that it does not change the property of a StructuralJoin being topmost or not. That is to say, no matter how we commute operators, the category that a StructuralJoin belongs to does not change. Since whether an operator should be commuted with a StructuralJoin is completely decided by the category of the StructuralJoin, which is unchanged, the order in which we commute operators with a StructuralJoin does not affect the final outcome.

Algorithm 6: Commuting Operators Using both Heuristics and Costs

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>for each StructuralJoin in the plan do</td>
</tr>
<tr>
<td>2</td>
<td>if StructuralJoin falls in the first or third category then</td>
</tr>
<tr>
<td>3</td>
<td>while its parent is a Select or NodeNav operator do</td>
</tr>
<tr>
<td>4</td>
<td>commute this StructuralJoin with its parent</td>
</tr>
<tr>
<td>5</td>
<td>end while</td>
</tr>
<tr>
<td>6</td>
<td>else</td>
</tr>
<tr>
<td>7</td>
<td>while it has Select or NodeNav child operators do</td>
</tr>
<tr>
<td>8</td>
<td>commute StructuralJoin with each child operator</td>
</tr>
<tr>
<td>9</td>
<td>end while</td>
</tr>
<tr>
<td>10</td>
<td>end if</td>
</tr>
<tr>
<td>11</td>
<td>end for</td>
</tr>
<tr>
<td>12</td>
<td>for each StructuralJoin in the plan do</td>
</tr>
<tr>
<td>13</td>
<td>for each input subplan of StructuralJoin do</td>
</tr>
<tr>
<td>14</td>
<td>commute operators within the input subplan according to their rank functions</td>
</tr>
<tr>
<td>15</td>
<td>end for</td>
</tr>
<tr>
<td>16</td>
<td>end for</td>
</tr>
</tbody>
</table>

Algorithm 6 shows the optimization using the operator-commuting rules. We perform the commuting in two steps. In the first step, we use the heuristics to commute StructuralJoin with Select or NodeNav operators (lines 1-11). Lemma 1 shows that the order in which we commute a Select or a NodeNav with StructuralJoin does not matter. We therefore traverse each StructuralJoin and commute
the Select or NodeNav operators with it. Since in the second step, we visit each input subplan of StructuralJoin operators. For each input subplan, we use the rank functions [19, 45] to commute between the Select and NodeNav operators (lines 12 - 16).

### 3.4 Using Rank Functions for Input Subplan Reordering

The problem of reordering input subplans of StructuralJoin bears some resemblance to the problem of ordering select and join operators [19, 45]. However, the operators [19, 45] considered to be reordered must have a consuming-producing relationship, i.e., the output of one select operator will be the input to another select operator. In contrast, the subplans in our scenario do not have such relationships. For example, in Figure 3.2, the output of the subplan containing Select$_{f=01609}$ is not sent to the subplan containing Select$_{c=\text{contains "frequent"}}$. We therefore extend the techniques in [19, 45] and derive a criterion, shown in Theorem 2, for deciding the optimal evaluation order of input subplans.

**Theorem 2** The cost of input subplans is minimal when they are evaluated in the ascending order of their rank functions as defined below:

\[
\text{rank}(\text{subplan}) = \frac{\sigma(\text{entryPlan(entryOp)}) \cdot \text{UnitCost(subplan)}}{1 - P_{\neq \emptyset}(\text{subplan})},
\]

The proof can be found in Appendix C. Intuitively, this criterion says, a subplan should be evaluated early if (1) its entry operator filters many of its input tuples, i.e., small $\sigma(\text{entryPlan(entryOp)})$, (2) it costs little, i.e., small UnitCost(subplan), and (3) it often does not generate any output each time when the StructuralJoin is invoked, i.e., small $P_{\neq \emptyset}(\text{subplan})$. 

3.5 Enumerative Search for One-time Optimization

In the previous two sections, we studied how to optimize the non-automaton part of a plan. We now address whether a pattern should be retrieved in the automaton or out of the automaton. In this section, we present an enumerative search algorithm which ensures: (1) all possible alternative plans are explored so that it guarantees to find the optimal plan and (2) an alternative is never explored twice.

Suppose the initial plan has \( n \) pattern retrieval operators. Our exhaustive search algorithm enumerates the combinations (i.e., subsets) of the \( n \) pattern retrieval operators. For each combination of operators, we change the modes of the operators in the initial plan and get an alternative plan. However, as stated in Lemma 2, certain combinations can lead to the generation of plans that are redundant. Such combinations are not explored by our exhaustive search algorithm.

**Lemma 2 Combinations Containing Operators with Pattern Dependency Relationship being Redundant:** Suppose \( \text{navOp}_1 \) and \( \text{navOp}_2 \) have pattern dependency relationship. They retrieve two patterns \( v = u/p_1 \) and \( y = x/p_2 \) where \( x \) is an element within \( v \). \( \text{navOp}_1 \) and \( \text{navOp}_2 \) can be either a TokenNav or a NodeNav type. They are not necessarily the same types. A combination containing both \( \text{navOp}_1 \) and \( \text{navOp}_2 \) produces the same plan as another combination that contains no operators with pattern dependency relationship.

**Proof 2** We distinguish between three cases: first, \( u/p_1 \) and \( x/p_2 \) are both retrieved in the automaton; second, \( u/p_1 \) and \( x/p_2 \) are both retrieved out of the automaton; third, \( u/p_1 \) is retrieved in the automaton while \( x/p_2 \) is retrieved out of the automaton. The fourth case, i.e., \( u/p_1 \) is retrieved out of the automaton...
while $x/p_2$ is retrieved in the automaton, is not supported in Raindrop. Because as mentioned in Section 2.4.3, if $u/p_1$ is retrieved out of the automaton, then its descendant pattern $u/p_1$ must be retrieved out of the automaton as well. We now prove that the combination in the first case is redundant. The proofs of the two other cases are similar and can be found in Appendix D.

**Case 1:** Suppose a plan contains a TokenNav$_{u,p_1}v$ and a TokenNav$_{x,p_2}y$ where $u/p_1$ is the ancestor pattern of $x/p_2$. Changing the modes of both means we pull out both $u/p_1$ and $x/p_2$. However, pulling out $u/p_1$ alone will make $x/p_2$ to be pulled out as a second effect. For example, in Figure 3.2, pulling out $a/seller$ requires $b/profile$ to be also pulled out. Therefore, this combination generates the same plan as the combination that contains TokenNav$_{u,p_1}v$ but not TokenNav$_{x,p_2}y$.

If a combination contains no pattern retrieval operators that have pattern dependency relationship with each other, we say this is a valid combination. Changing the modes in a valid combination must uniquely lead to a plan, regardless of the order in which we change the modes of the operators in it. Lemma 3 states this order insensitive property of a valid combination.

**Lemma 3 Combinations being Order Insensitive:** If two pattern retrieval operators navOp$_1$ and navOp$_2$ have no pattern-dependency relationship, then regardless of the order in which we change the modes of navOp$_1$ and navOp$_2$, the two plans derived contain the same operators.

**Proof 3** We distinguish between three cases the same as those used for proving Lemma 2. We prove the first case, i.e., given two operators TokenNav$_1$ and
3.5. ENUMERATIVE SEARCH FOR ONE-TIME OPTIMIZATION

TokenNav₂, no matter in what order we change their modes, we get the same plans. The proof for the other two cases can be found in Appendix E.

Case 1: Suppose TokenNav₁ = TokenNavₘₐₓₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜ১

First, TokenNavₘₐₓₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜₜ১ and Extractₚₜₜₜₜₜₜₜₜ১ are rewritten into NodeNavₘₐₓₜₜ১. Second, if before the rewriting there exists no Extract operator that extracts $u$, then an Extract operator that extracts $u$ will be introduced to the plan after the rewriting. Third, the descendant patterns of $u/p$ that are retrieved in the automaton will be pulled out. Fourth, if there exists no other operator in the format of TokenNav₁, TokenNavₘₐₓₜ১, $u/p$ but there exists a StructuralJoin$_p$ before the rewriting, this StructuralJoin$_p$ is eliminated after the rewriting.

Later, if we change the mode of TokenNav₂, we have the below observations:

1). Mode change of TokenNav₂ will introduce neither a TokenNavₘₐₓₜ১ nor a Extractₚ১. It will not eliminate an Extract. Neither will it introduce a StructuralJoin operator. Therefore it will not cancel out the first, second and fourth changes resulted from the mode change of TokenNav₁.

2). Since TokenNav₂ does not have a pattern dependency relationship with TokenNav₁, mode change of TokenNav₂ will not affect those operators whose modes have been changed as a secondary effect of the pull-out of $u/p$. That is to say, it will not cancel out the third change resulted from the mode change of TokenNav₁.

In summary, a mode change of TokenNav₂ that occurs after the mode change of TokenNav₁ does not cancel any change that has been made. Therefore the
order in which we change the modes of $\text{TokenNav}_1$ and $\text{TokenNav}_2$ does not matter.

**Algorithm 7 Exhaustive Search**

ExhaustOpt($curPlan$, $navsToBeTried$)

Input: $curPlan$ - a current plan, will be set to the initial plan when the algorithm is first called;
$navsToBeTried$ - a list of pattern retrieval operators eligible for mode changes;

Output: the best plan in the search space

1: Plan $bestPlan = curPlan$.
2: int $n =$ number of pattern retrieval operators in $navsToBeTried$;
3: for (int $i = 1; i \leq n; i++$) do
4:   NavOp $curNavOp = i^{th}$ operator in $navsToBeTried$;
5:   Plan $newPlan =$ copy of $curPlan$;
6:   Change mode of $curNavOp$ in $newPlan$;
7:   Optimize $newPlan$ using operator-commuting rules (see Algorithm 6);
8:   Optimize $newPlan$ using input-subplan-reordering rules;
9:   List $newNavsToBeTried$;
10: for (int $j = i + 1; j \leq n; j++$) do
11:   NavOp $newNavOp = j^{th}$ operator in $navsToBeTried$;
12:   if $newNavOp$ and $curNavOp$ have no pattern dependency relationship then
13:      add $newNavOp$ into $newNavsToBeTried$;
14:   end if
15: end for
16: $curBestPlan = \text{ExhaustOpt}(newPlan, newNavsToBeTried)$;
17: if $curBestPlan$ costs less than $bestPlan$ then
18:   $bestPlan = curBestPlan$.
19: end if
20: end for
21: return $bestPlan$.

Algorithm 7 utilizes Lemmas 2 and 3 to search through the solution space.
The algorithm $\text{ExhaustOpt}$ takes two input parameters, namely, a plan and a list of pattern retrieval operators. The first time $\text{ExhaustOpt}$ is called, $curPlan$ is
3.5. ENUMERATIVE SEARCH FOR ONE-TIME OPTIMIZATION

the initial plan and navsToBeTried contains all the pattern retrieval operators in the initial plan. Suppose there are \( n \) operators in navsToBeTried. For each operator \( navOp_i \) (\( 1 \leq i \leq n \)) in the navsToBeTried list, we make a copy of curPlan, change the mode of navOp in the plan copy and then get a new plan (lines 4 - 6). We will get \( n \) new plans. We recursively apply exhaustiveSearch with the input parameters newPlan and newNavsToBeTried (line 16). For a plan newPlan that results from the mode change of navOp, we make sure that newNavsToBeTried does not contain any operators that have dependency relationship with navOp (lines 12 - 13), because changing the modes of such an operator and navOp is forbidden by the invalid combination lemma.

We now illustrate this algorithm ensures that no alternative plan is missed. Given an arbitrary plan \( P_{\text{any}} \) that can be derive by changing modes of a sublist of the navsToBeTried in the initial plan, it must be explored by ExhaustOpt. We denoted the sublist as \( S \). Assume \( S = \{ navOp_k, navOp_{k+1}, \ldots \} \) (\( 1 \leq k < \) number of pattern retrieval operators in navsToBeTried). When ExhaustOpt is called the first time, among \( n \) new plans, one results from the mode change of navOp_k. When ExhaustOpt is recursively called on this new plan, it will change the mode of navOp_{k+1}. The process continues. When ExhaustOpt is called the \( |S| \)th time (\( |S| \) denotes the number of operators in \( S \)), \( P_{\text{any}} \) must be generated.

Also, the process mentioned above is the only way that ExhaustOpt can generate \( P_{\text{any}} \). Whenever we change the mode of a navOp in the current plan and get a new plan, only operators occurring after navOp in the navsToBeTried list are added into the newNavsToBeTried list (see line 10 where \( j \) starts from \( i + 1 \)). \( P_{\text{any}} \) can be only generated when the exhaustive applies token-or-node rewrite rule in the order of navOp_k, navOp_{k+1} and so on. Therefore \( P_{\text{any}} \) is never explored
3.6 Greedy Search for One-time Optimization

For a query with $n$ patterns, the search space can have up to $\sum_{i=0}^{n} C_i^n = 2^n$ alternative plans. We say “up to” because some combinations are invalid and thus excluded. Finding an optimal plan obviously will be time-consuming. In this section, we present a greedy search algorithm that aims to quickly find a good but not necessarily optimal plan.

3.6.1 Baseline Greedy Search

Figure 3.12 intuitively depicts how Greedy search works. The initial plan shown as $P$ in Figure 3.12 has a set of pattern retrieval operators denoted as $S_0$. For each pattern retrieval operator in $S_0$, we change its mode and get a new plan, denoted as $P_1, ..., P_n$ respectively. We use operator-commuting and input-subplan-reordering rules to further optimize the new plans (the circles on $P_1, ..., P_n$ in Figure 3.12 denote such optimization). If the cheapest plan among the $n$ new plans is also cheaper than the initial plan, we then select this cheapest plan as the new current plan. For example, in Figure 3.12, $P_1$, which results from the mode change of $navOp_1$, is selected after the first iteration (the highlighted node represents a new current plan).

With the newly selected plan, we begin the next iteration. In this iteration, we again change the mode of a set of operators, denoted as $S_1$, where $S_1 = S_0 - navOp_1$ — operators that have pattern dependency relationship with $navOp_1$. Similar to the first iteration, we optimize, cost and compare the new plans. We
3.6. GREEDY SEARCH FOR ONE-TIME OPTIMIZATION

then get a new current plan $P_{12}$ in Figure 3.12. The iterations continue until no new plan is found to be cheaper than the current plan, i.e., the best plan found so far. Algorithm 8 shows the pseudocode for this search process. We call this algorithm $GreedyOpt$.

![Figure 3.12: Greedy-based Search](image)

We now compute the upper bound on the number of alternative plans explored by the $GreedyOpt$ algorithm. In the first iteration, $GreedyOpt$ explore $n$ alternatives plans. In the second iteration, $GreedyOpt$ explore at most $n - 1$ alternative plans. After at most $n$ iterations, the process terminates. Therefore $GreedyOpt$ explore at most $\sum_{i=1}^{n} i = n(n + 1)/2$ alternative plans.

### 3.6.2 Expediting Cost Estimate

In the section, we propose techniques to expedite the $GreedyOpt$ algorithm. These techniques reduce the time spent on processing an alternative plan, more specifically, costing an alternative plan. When we apply a mode change and get a new plan, we need to cost this new plan. In a naive approach, we recompute the cost from scratch. In contrast, we can analyze what parts of plan are affected by the...
Algorithm 8 Greedy Search in an One-time Optimization Scenario

GreedyOpt(curPlan, navsToBeTried)
Input: curPlan - a current plan, will be set to the initial plan when the algorithm is first called;

navsToBeTried - a list of pattern retrieval operators eligible for mode changes;
Output: the best plan among the plans explored

1: Plan bestPlan = curPlan;
2: for each operator navOp in navsToBeTried do
3:   Plan newPlan = copy of curPlan;
4:   Change mode of navOp in newPlan;
5:   Optimize newPlan using operator-commuting rules;
6:   Optimize newPlan using input-subplan-reordering rules;
7:   if newPlan costs less than bestPlan then
8:      bestPlan = newPlan
9:   end if
10: end for
11: if bestPlan != curPlan then
12:   let navOp_i denotes the operator whose mode change leads to bestPlan;
13:   navsToBeTried = navsToBeTried - navOp_i - all operators that have pattern dependency relationship with navOp_i;
14:   return GreedyOpt(bestPlan, navsToBeTried);
15: else
16:   return curPlan.
17: end if
3.6. GREEDY SEARCH FOR ONE-TIME OPTIMIZATION

mode change and avoid recomputing. We propose two techniques, i.e., incremental cost estimate and detection of same cost change.

**Incremental Cost Estimate.**

We first define several concepts needed in our analysis.

**Definition 4** For a NavOp \(u,p\) \(v\), we call StructuralJoin \(u\) its context StructuralJoin because StructuralJoin \(u\) joins on the context element \(u\) of NavOp \(u,p\) \(v\).

**Definition 5** The heuristics in Section 3.1 impose that NodeNav \(u,p\) \(v\) cannot be moved above a StructuralJoin \(v\) that can output duplicates of bindings of \(v\) or is the topmost StructuralJoin. We call this StructuralJoin a confining StructuralJoin of NavOp \(u,p\) \(v\).

The confining StructuralJoin confines how far the NavOp \(u,p\) \(v\) operator itself (when NavOp \(u,p\) \(v\) is a NodeNav) or the TokenNav operator rewritten from NavOp \(u,p\) \(v\) (when NavOp \(u,p\) \(v\) is a TokenNav) can be moved up.

**Definition 6** We define a function moveScope(navOp) to denote a set of StructuralJoins. The set consists of all the StructuralJoins between the context and confining StructuralJoins of navOp, including the context and confining StructuralJoins.

**Example 13** In Figure 3.2, for TokenNav \(a,/seller\) \(b\), StructuralJoin \(a\) is its context StructuralJoin. If we change the mode of TokenNav \(a,/seller\) \(b\), the resulting NodeNav \(a,/seller\) \(b\) cannot be moved above StructuralJoin \(a\) because there exists an ExtractUnnest \(a\) \(c\) in the plan. StructuralJoin \(a\) is then also TokenNav \(a,/seller\) \(b\)'s confining StructuralJoin. moveScope(TokenNav \(a,/seller\) \(b\)) thus is \{StructuralJoin \(a\}\}. 
Suppose we change the mode of a $\text{TokenNav}_{u,p}^v$ operator in the current plan $P_{\text{current}}$ and get $P_{new}$. We use a boolean value $\text{isIntroduced}$ to denote whether an operator in the form of $\text{Extract}_{t}^u$ is introduced into $P_{new}$ because of this change. We then have Equation 7.

**Equation 7** Cost change from a pattern pull-out

\[
\text{Cost change from a pattern pull-out} = \text{Cost}(P_{new}) - \text{Cost}(P_{\text{current}}) \\
= \text{automaton cost in } P_{new} - \text{automaton cost in } P_{\text{current}} \\
+ \text{automaton-outside cost in } P_{new} - \text{automaton-outside cost in } P_{\text{current}} \\
= \text{Cost}(\text{Extract}_{t}^u) \times \text{isIntroduced} - \text{Cost}(\text{TokenNav}_{u,p}^v) \\
+ \sum_{s_j \in \text{moveScope}(\text{TokenNav}_{u,p}^v)} \text{cost of input subplans of } s_j \text{ in } P_{new} \\
- \sum_{s_j \in \text{moveScope}(\text{TokenNav}_{u,p}^v)} \text{cost of input subplans of } s_j \text{ in } P_{\text{current}}
\]  

According to Equation 6 in Section 3.2.3, a $\text{TokenNav}_{u,p}^v$ operator costs the same in every alternative plan where it appears. However, changing $\text{TokenNav}_{u,p}^v$ to $\text{NodeNav}_{u,p}^v$ can introduce a new $\text{Extract}_{t}^u$ operator if $u$ was not extracted in the current plan. Therefore, we expand Expression (1) into Expression (3) in Equation 6. Also, since $\text{NodeNav}_{u,p}^v$ cannot be placed above the confining $\text{StructuralJoin}$, the mode change does not affect the operators beneath the confining $\text{StructuralJoin}$. We thus expand Expression (2) into Expression (4) in Equation 7. We can use Equation 7 to compute the cost change from $P_{\text{current}}$ to $P_{new}$ when we pull out a pattern from the automaton.

Equation 8 gives the cost change that would result from a push-in of a pattern into the automaton. Equation 8 is the reverse of Equation 7. $\text{isEliminated}$ is a boolean value that indicates whether an operator in the form of $\text{Extract}_{t}^u$ is eliminated from $P_{\text{current}}$ because of this change.
3.6. GREEDY SEARCH FOR ONE-TIME OPTIMIZATION

Equation 8  Cost change from a pattern push-in

\[ \text{Cost change from a pattern push-in} = \text{Cost}(P_{\text{new}}) - \text{Cost}(P_{\text{current}}) \]

\[ = \text{Cost}(\text{TokenNav}_{u,p}^v) - \text{Cost}(\text{Extract}_{\text{isEliminated}}) + \sum_{s_j \in \text{moveScope}(\text{TokenNav}_{u,p}^v)} \text{cost of input subplans of } s_j \text{ in } P_{\text{new}} \]

\[ - \sum_{s_j \in \text{moveScope}(\text{TokenNav}_{u,p}^v)} \text{cost of input subplans of } s_j \text{ in } P_{\text{current}} \]

Detection of Same Cost Change.

From Equations 7 and 8, we can derive Theorem 3.

Theorem 3  Given two pattern retrieval operators navOp\(_1\) and navOp\(_2\), if \(\text{moveScope}(\text{navOp}_1) \cap \text{moveScope}(\text{navOp}_2) = \emptyset\), then the cost change resulted from the mode change of navOp\(_1\) in a plan is independent from the mode change of navOp\(_2\) in this plan.

The proof of Theorem 3 can be found in Appendix F. Figure 3.13 shows how to utilize Theorem 3. Given a plan \(P_1\), we get two plans \(P_2\) and \(P_3\) by changing the modes of navOp\(_1\) and navOp\(_2\) in \(P_1\) respectively. Suppose we now change the mode of navOp\(_1\) in \(P_3\) and get a new plan \(P_4\). If \(\text{moveScope}(\text{navOp}_1) \cap \text{moveScope}(\text{navOp}_2) = \emptyset\), we then know \(\text{Cost}(P_4) - \text{Cost}(P_3)\) is the same as \(\text{Cost}(P_2) - \text{Cost}(P_1)\). We can simply compute \(\text{Cost}(P_4)\) as \(\text{Cost}(P_3) + \text{Cost}(P_2) - \text{Cost}(P_1)\).

![Figure 3.13: Detection of Same Cost Change: Is Cost(P\(_4\)) - Cost(P\(_3\)) = Cost(P\(_2\)) - Cost(P\(_1\))?](image)
Example 14 Suppose we have a plan in Figure 3.14 (a). This plan corresponds to $P_1$ in Figure 3.13. We pull out each of the four $TokenNav$ operators respectively and get four new plans. Assume we chose the plan after the pull-out of $TokenNav_{a,p}^b{}^e$, shown in Figure 3.14 (b), as the new current plan. This new current plan corresponds to $P_3$ in Figure 3.13. The part in Figure 3.14 (b) that is different from Figure 3.14 (a) is highlighted. To make the next move, we now need to pull out $TokenNav_{a,p}^b{}^c$ (we do not consider the pull-out of $TokenNav_{a,p}^b{}^d$ because it has pattern dependency relationship with $TokenNav_{a,p}^b{}^d$). Thereafter, we need to estimate the costs of the two new plans.

1). For $TokenNav_{a,p}^b{}^e$, $moveScope(TokenNav_{a,p}^b{}^d) \cap moveScope(TokenNav_{a,p}^b{}^e) = \{StructuralJoin_b\} \cap \{StructuralJoin_{5a}\} = \emptyset$. Therefore the two cost changes that the pull-out of $TokenNav_{a,p}^b{}^e$ in Figures 3.14 (a) and (b) cause respectively are the same. We can reuse the estimate of the cost change from the last time.

2). In contrast, for $TokenNav_{a,p}^b{}^c$, $moveScope(TokenNav_{a,p}^b{}^d) \cap moveScope(TokenNav_{a,p}^b{}^c) = \{StructuralJoin_b\} \cap \{StructuralJoin_{5a}{}^b\} = \{StructuralJoin_{5b}\} \neq \emptyset$. The two cost changes that the pull-out of $TokenNav_{a,p}^b{}^c$ in Figures 3.14 (a) and (b) cause respectively are different. We cannot reuse the estimate of cost change from last time.

Summary

When we get a new plan, we first apply the technique of “detection of same cost change”. If we find out that the cost change is not the same as estimated last time,
3.6. GREEDY SEARCH FOR ONE-TIME OPTIMIZATION

Figure 3.14: Reuse Cost Estimate for Mode Changes of Patterns in Figure 3.14 (a)

we then apply the technique of “incremental cost estimate”. 
3.7 Greedy Search with Pruning for Continuous Optimization

If the environment fluctuates, we have to optimize more frequently than in the one-time optimization scenario (see Section 1.3.2 in Section 1). Correspondingly, we need to find a good plan even more quickly. The plan search time is decided by two factors, i.e., number of alternative plans explored and the time spent on each alternative plan. The GreedyOpt algorithm has reduced the plan search time of the ExhaustOpt algorithm by reducing the two factors, i.e., using a Greedy search strategy and expediting costing of a plan respectively. Within the current search space that is delimited by the three rewrite rules, it is hard to further reduce the two factors in the GreedyOpt. We therefore consider dropping some rewrite rules to shrink the search space.

Among the three types of rewrite rules introduced in Section 3.1, token-or-node mode change and operator-commuting rules are more likely to affect the plan performance than the input-subplan-reordering rule. Token-or-node mode change rules enable the plans to benefit from pulling out (resp. pushing in) pattern retrieval with high (resp. low) selectivity. Operator-commuting rules enable the plans to benefit from executing operators with high selectivity after others. In contrast, plans benefit from input subplan reordering only if an input subplan of a $StructuralJoin_{v}$ does not generate output within a binding of $v$, i.e., $P_{\varphi \theta}(subplan) = 0$. This is a rather strict requirement. Therefore, we drop the input-subplan-reordering rule.

Dropping the input-subplan-reordering rule simply means we do not change the left to right order of the input subplans of a $StructuralJoin$. The execution manner
3.7. GREEDY SEARCH WITH PRUNING FOR CONTINUOUS OPTIMIZATION

of StructuralJoin given in Algorithm 5 remains unchanged. StructuralJoin still first checks whether all entry operators have generated output; if yes, it then evaluates the input subplan from left to right and terminates if any input subplan does not generate output. Therefore, the cost model of input subplans does not change.

Among the three heuristics for operator commuting, one heuristic is proposed in order to provide optimization opportunities for input subplan reordering. The heuristic says that we should place Select or NodeNav operators above a StructuralJoin that is not a topmost StructuralJoin and does not output tuples with duplicate bindings of $v$. Even though we drop the input-subplan-reordering optimization, we still keep the heuristics for two reasons.

First, a Select or NodeNav operator is still less likely to be evaluated after being commuted with its parent StructuralJoin operator. Dropping the input-subplan-reordering optimization only means the benefit we get from this commuting is not maximal. Second, the side effect of placing Select or NodeNav above StructuralJoin is small so that it would not offset the benefit of an even sub-optimal input subplan order. When we commute a Select or NodeNav with its parent StructuralJoin, the cost of StructuralJoin increases since the Select or NodeNav could otherwise filter out the input to StructuralJoin without the commuting. However, we only place a Select or NodeNav above a StructuralJoin that does not output duplicate bindings of $v$. This means, no operator in the plan is in the format of ExtractUnnest$_{v}$, Nest$_{v}$, or NavNest$_{v,p}$. As a result, each time when an end tag of a binding of $v$ is encountered, StructuralJoin has at most one tuple from each input operator (e.g., StructuralJoin$_{b}$ consumes at most one tuple from ExtractNest$_{b}//profile$ for each binding of $b$). Therefore
3.7. GREEDY SEARCH WITH PRUNING FOR CONTINUOUS OPTIMIZATION

there is not much space to further reduce the cost of this $StructuralJoin_{sv}$. In summary, dropping the input-subplan-reordering optimization does not affect the operator-commuting optimization.

For the greedy algorithm, we only need to make a slight change in order for it to apply to our new scenario. We remove the input-subplan-ordering optimization on a plan, i.e., remove line 5 in Algorithm 8. For the new greedy algorithm, we further propose a technique for pruning the alternative plans, i.e., reducing the number of alternative plans to explore. The greedy algorithm with the pruning technique guarantees to find the same plan as the greedy algorithm without pruning.

3.7.1 Basic Ideas of Pruning

Suppose we can estimate a lower bound of the cost changes from the mode changes of $navOp$ in any plans that contain $navOp$ and have been optimized by operator-commuting rule, where cost change is defined as (cost of the plan after mode change - cost of the plan). If this lower bound is larger than 0, it means that for any plan, $(cost of the plan after the mode change of navOp - cost of the plan) > 0$. In other words, mode change of this navOp in any plans always leads to a worse plan. We can then safely exclude the mode change of navOp in any plans.

The challenge is then how to compute the lower bound. We want the computation to satisfy two properties. First, it should be quick; otherwise the computation overhead may offset the benefits of saving time in exploring the alternative plans. Second, the lower bound computed should be “tight”. For example, an extremely relaxed lower bound “negative infinite” (cost of new plan - cost of current plan must always be greater than negative infinite) will not exclude anything. Only a lower bound that is greater than 0 can help pruning alternative plans. These two
properties usually have a negative correlation, i.e., we usually need to spend more
time to compute a tighter lower bound. We have to strike a balance between the
time spent on computing the lower bound and the quality of the lower bound.

3.7.2 Pruning Plans Derived from Mode Change of TokenNav Operators

We first consider the case in which \( navOp \) is a \( TokenNav_{u,p,v} \) whose \( v \) is not selected by any \( Select \) or navigated into by any \( NodeNav \). When we pull out such a \( TokenNav_{u,p,v} \) in a current plan \( P_{current} \) and get a new plan \( P_{new} \), then no other operator would have to be moved so that they are still placed above \( TokenNav_{u,p,v} \). This leads to Equation 9.

**Equation 9 Cost change of changing mode of TokenNav_{u,p,v} with \( v \) not being consumed by other operators:**

\[
\text{Cost}(P_{new}) - \text{Cost}(P_{current}) = \text{automaton cost in } P_{new} - \text{automaton cost in } P_{current} \tag{1}
\]

\[
+ \text{automaton-outside cost in } P_{new} - \text{automaton-outside cost in } P_{current} \tag{2}
\]

\[
= \text{Cost}(\text{Extract}_{u}v) \ast \text{isIntroduced} - \text{Cost}(\text{TokenNav}_{u,p,v}) \tag{3}
\]

\[
+ \text{Cost}(\text{NodeNav}_{u,p,v}) \tag{4}
\]

\[
- \text{Cost}(\text{StructuralJoin}_{u} \text{ in } P_{current}) \ast \text{isEliminated} \tag{5}
\]

\[
+ \text{cost of rest automaton-outside operators in } P_{new} - \text{cost of rest automaton-outside operators in } P_{current} \tag{6}
\]

Expressions (1), (2) and (3) are the same as Equation 7. Expression (1) is expanded into Expression (3). Expression (2) is expanded into Expressions (4) and (5). For Expression (4), depending on \( NodeNav_{u,p,v} \)’s descendent operators,
3.7. GREEDY SEARCH WITH PRUNING FOR CONTINUOUS OPTIMIZATION

Cost(NodeNavₘₚᵥ) can vary in different current plans. Cost(NodeNavₘₚᵥ) is minimal when NodeNavₘₚᵥ is executed as late as possible. In such cases it consumes the least input and thus costs the least. We denote this minimal cost as \( \min(Cost(NodeNavₘₚᵥ)) \). Therefore Expression (4) \( > \min(Cost(NodeNavₘₚᵥ)) \).

We now analyze the lower bound of Expression (5) in Equation 9. Changing the mode of TokenNavₘₚᵥ can lead to the elimination of StructuralJoinₘ. This can happen in only one case. That is, when StructuralJoinₘ in the current plan has only two input subplans according to the mode change with introducing/eliminating StructuralJoin rewrite rule in Figure 2.6. Therefore Expression (5) \( > -Cost(StructuralJoinₘ) \).

Except the possibly eliminated StructuralJoinₘ, all the other automaton-outside operators in \( P_{current} \) remain in \( P_{new} \). Also, the rank of each such automaton-outside operator \( op \), i.e., \( \sigma(op)^{-1} \cdot Unittest(op) \), is completely decided by the \( op \) itself. It is not changed by the newly created NodeNavₘₚᵥ. Therefore commuting these automaton-outside operators with each other is not needed. However, rewriting TokenNavₘₚᵥ to NodeNavₘₚᵥ can increase the cost of those automaton-outside operators which are executed after TokenNavₘₚᵥ in \( P_{current} \) but are executed before NodeNavₘₚᵥ in \( P_{new} \). Therefore, Expression (6) \( \geq 0 \).

Based on the above discussion, we have Equation 7 \( > -Cost(TokenNavₘₚᵥ) + \min(Cost(NodeNavₘₚᵥ)) - Cost(StructuralJoinₘ) \). Correspondingly, we have Pruning Rule 1.

**Pruning Rule 1** Given a pattern \( \$v = \$/p \) where \$v is not further selected by Select operators or navigated into by NodeNav operators, if \( \min(Cost(NodeNavₘₚᵥ)) - Cost(TokenNavₘₚᵥ) - Cost(StructuralJoinₘ) > 0 \), we do not consider
3.7. GREEDY SEARCH WITH PRUNING FOR CONTINUOUS OPTIMIZATION

3.7.3 Discussion on Pruning Other Pattern Retrieval Operators

We now discuss $TokenNav_{u,p}v$ operators whose bindings of $v$ are further selected on or navigated into. To get the lower bound of (cost of $P_{new}$ - cost of $P_{current}$) for $TokenNav_{u,p}v$, we have to estimate the lower bound of those operators that consume $v$ in $P_{new}$ by assuming they are executed as late as possible; and the upper bound of these operators in $P_{current}$ by assuming they are executed as early as possible. Doing this can be quite time-consuming. We therefore do not consider pruning by bounding the cost of the mode change of $TokenNav_{u,p}v$ whose $v$ is further consumed.

The cost change that results from the mode change of $NodeNav_{u,p}v$ whose bindings of $v$ are not consumed by other operators is the reverse of Equation 9.

**Equation 10** Cost change of changing mode of $NodeNav_{u,p}v$ with $v$ not being consumed by other operators:

\[
\text{Cost}(P_{new}) - \text{Cost}(P_{current}) = \\
\text{Cost}(TokenNav_{u,p}v) - \text{Cost}(Extract_{u}v) \ast isIntroduced \quad (1) \\
- \text{Cost}(NodeNav_{u,p}v) \quad (2) \\
+ \text{Cost}(StructuralJoin_{u}) \text{ in } P_{current} \ast isEliminated \quad (3) \\
- \text{cost of rest automaton-outside operators in } P_{new} + \text{cost of rest automaton-outside operators in } P_{current} \quad (4)
\]

Since Expressions (3) and (6) in Equation 9 both are greater than some constants, Expressions (1) and (4) in Equation 10 are then less than these constants. It is difficult to get a lower bound for this cost change. We therefore do not develop
3.8. EMBEDDING STATISTICS COLLECTION INTO PLAN EXECUTION

a pruning rule for bounding the cost change caused by pushing in a node pattern retrieval.

3.7.4 Summary

Algorithm 9, which is called greedyPruneOpt, shows the greedy search with pruning for the continuous optimization scenario. Each time when we start the optimization, we call greedyPruneOpt with three parameters, curPlan which is the currently running plan, navsToBeTried which is a list of pattern retrieval operators in curPlan, and a boolean value TRUE to indicate that this is the first iteration of the optimization on curPlan. During the first iteration of the optimization, we apply the technique of “pruning by bounding cost change” (lines 1 - 8 in Algorithm 9). We bound the cost change for each TokenNav whose pattern is not further consumed in the plan. Those operators whose lower bound is greater than 0 are excluded from mode changes. With the rest pattern retrieval operators, we then apply greedy search as before (lines 9 - 16 in Algorithm 9).

3.8 Embedding Statistics Collection into Plan Execution

We now analyze what statistics need to be collected to estimate the costs of the plans. Tables 3.1 and 3.2 in Section 3.2.3 contain the parameters needed for costing the automaton. Some of the parameters, such as \( C_{\text{nonEmp}} \), \( C_{\text{emp}} \), \( C_{\text{visit}} \) and \( C_{\text{bicartesian}} \) are constants. We can determine their values off-line, i.e., before the data comes in. The other parameters, namely, \( C_{\text{extract}}(q) \), \( n_{\text{active}}(q) \), \( n_{\text{start}} \), \( n_{p[i]} \) and \( w_{p[i]} \) vary in different data and need to be collected on-line, i.e., as the data comes in. The parameter \( C_{\text{backtrack}} \) in Table 3.2 is only used for analysis, but not needed in Equation 6 which computes the cost of a TokenNav operator. We therefore do not collect it.
Algorithm 9 Greedy Search with Pruning in a Continuous Optimization Scenario

GreedyPruneOpt(curPlan, navsToBeTried, isInitial)

Input: curPlan - a current plan, will be set to the initial plan when the algorithm is first called;
navsToBeTried - a list of pattern retrieval operators eligible for mode changes;
isInitial - a boolean indicating whether curPlan is the initial plan;

Output: the best plan among the plans explored

1: if isInitial then
2:   for each operator navOp in navsToBeTried that satisfies: (1) navOp is a TokenNav and (2) the pattern retrieved
3:     by navOp is not further consumed do
4:       double lowerBound = estimate lower bound of the cost cut of mode change on tokenNavOp;
5:       if lowerBound > 0 then
6:         remove tokenNavOp from navsToBeTried
7:     end if
8:   end for
9:   ... (same as lines 1 - 7 in Algorithm 8)
10:  if bestNewPlan costs less than curPlan then
11:    let navOp_i denote the operator whose mode change leads to bestNewPlan;
12:    navsToBeTried = navsToBeTried − navOp_i − all operators that have pattern dependency relationship with NavOp_i;
13:    return GreedyPruneOpt(bestNewPlan, navsToBeTried, FALSE);
14:  else
15:    return curPlan.
16: end if
3.8. Embedding Statistics Collection into Plan Execution

comes in. Also, \( \sigma(op) \), \( P_{op} \neq \emptyset \) and \( UnitCost(op) \) are required in Equation 3 in Section 3.2.1 for costing the automaton-outside operators.

Some parameters can be derived from the others. For example, \( n_{p[i]} \) and \( w_{p[i]} \) are used to estimate the cost of a NodeNav\( _{u,p}$v while \( n_{active}(q) \) is used to estimate the cost of a TokenNav\( _{u,p}$v, \( n_{p[i]} \times w_{p[i]} \) gives the number of children of the bindings of \( p[i] \) (i.e., the \( i^{th} \) navigation step on \( p \)) in a binding of \( $u \). Suppose states \( q \) and \( q' \) in the automaton are activated by bindings of \( p[i] \) and binding of \( $u \) respectively. \( n_{active}(q) \) is the number of children of bindings of \( p[i] \) in a bottom input element (see Table 3.2). We then have, \( \frac{n_{active}(q)}{\text{number of bindings of } $u \text{ in a bottom input element}} = \sigma_{p[i]} \times w_{p[i]} \). Therefore we need only collect either \( n_{active}(q) \) when \( $u/p \) is retrieved in the automaton; or \( n_{p[i]} \) and \( w_{p[i]} \) when \( $u/p \) is retrieved out of the automaton.

We now briefly introduce how we collect each required parameter:

1. \( n_{active}(q) \): \( n_{active}(q) \) is the number of times that stack top contains a state \( q \) when a start tag arrives. For each state \( q \) in the automaton, we maintain a counter denoted as \( activeCounter(q) \). Each time when a start tag arrives, this counter of each state at the top of the stack is incremented by 1. Also, for a state that corresponds to the start of a path (e.g., \( q_2 \) in the automaton in Figure 3.2), we associate it with a second counter denoted as \( reachCounter(q) \). \( reachCounter(q) \) is incremented by 1 each time when \( q \) is pushed into the stack. For example, in Figure 3.2, when a start tag of a descendant of \( bidder \) elements arrives, the stack top always contains a \( q_8 \) so that \( activeCount(q_8) \) is incremented. When a <auction> arrives, it activates \( q_2 \) and \( reachCount(q_2) \) is incremented. \( n_{active}(q_8) \), i.e., the number of descendants of \( bidder \) in an auction, is then equivalent to \( \frac{activeCount(q_8)}{reachCount(q_2)} \).

2. \( C_{extract}(q) \): To find out the cost of storing elements whose start tags activate
3.9. RUN-TIME PLAN MIGRATION

In the compile time optimization, plan migration is not needed. We optimize, get a plan and simply run it. In the run time optimization in our scenario, we optimize a currently running plan, get a new plan (if any), and then have to migrate the current plan to this new plan. Two problems arise. First, how to change the current state \(q\), we maintain a storing cost counter denoted as \(storeCount(q)\). Also, the storage manager maintains a list. We can add a storing cost counter into the list or remove one from the list. Each time when \(q\) is activated, we add \(storeCount(q)\) into the list. Whenever the storage manager stores a token, it traverses this list. For each storing cost counter in the list, the storage manager increments it by the length of the token. Later, when \(q\) is popped off the stack, we remove its storing cost counter from the list. At this time, the value of \(storeCount(q)\) is the length of the element that activates \(q\).

3. \(P_{op \neq \emptyset}\): Assume \(StructuralJoin_{v}\) is \(op\)'s nearest ancestor \(StructuralJoin\). \(notEmptyCount(op)\) is the probability of \(op\) generating some output within a binding of \(v\). We associate \(op\) with a counter, denoted as \(notEmptyCount(op)\). Each time when \(StructuralJoin_{v}\) invokes \(op\) as an end tag of a binding of \(v\) arrives, \(notEmptyCount(op)\) is incremented by 1 if \(op\) generates some output. Suppose bindings of \(v\) activate automaton state \(q\), then at any time when a binding of \(v\) has been finished processing, \(\frac{notEmptyCount(op)}{activateCount(q)}\) gives \(P_{op \neq \emptyset}\).

The collection of \(\sigma(op)\) (selectivity of an operator), \(UnitCost(op)\) (cost of processing one input tuple) and \(n_{start}\) (number of start tags in a bottom input element) is rather straightforward. We skip the discussion here.
place to the new plan. This process must be efficient, especially in the continuous optimization scenario since plan change happens from time to time. Second, we need to determine when to change the current plan to the new plan. The migration should take place as soon as possible so that we can benefit from the new plan as early as possible. We now address these two aspects in Sections 3.9.1 and 3.9.2 respectively.

3.9.1 Incremental Change of Automaton

The search algorithm returns a new query plan. However this plan is not ready for execution. We must traverse the TokenNav operators in the new plan and construct an automaton out of it. For example, the plan search algorithm may return the top query plan in Figure 3.2 as the new plan found. We then need to construct the bottom automaton in Figure 3.2 before the plan can be executed.

We actually do not have to reconstruct the automaton from scratch. We can modify the automaton for the currently running plan and reuse it for the new plan. Besides a new plan, the search algorithm returns a list of NodeNav and TokenNav operators in the current plan whose modes have been changed. For each operator in the list, if the operator is a $\text{TokenNav}_{u,v}$, we remove the states that encode $p$ in the current automaton; if a $\text{NodeNav}_{u,v}$ has been pushed in, we add states to the current automaton to encode $p$.

For example, suppose we want to migrate the currently running plan in Figure 3.2 to the new plan in Figure 3.6. The mode of $\text{TokenNav}_{\text{buyer}}$, bidder, $e$ in the current plan is changed. Correspondingly, as shown in Figure 3.15, we remove the transition from $q_2$ to $q_4$ in the automaton in Figure 3.2. We still maintain the disconnected sub-automaton composed of states $q_4$, $q_5$ and $q_6$ which encodes
To $TokenNav_{sa./seller}$b. Next time, if the mode of $NodeNav_{sa./seller}$b in Figure 3.6 is changed, we can simply add the sub-automaton encoding $TokenNav_{sa./seller}$b in, namely, we add the transition from $q_2$ to $q_4$ without creating any new states.

![Diagram](https://via.placeholder.com/150)

**Figure 3.15: Incremental Change of Automaton for Migrating from Plan in Figure 3.2 to Plan in Figure 3.6**

We now have the automaton for the new plan. The next thing to do is then to associate the automaton with the operators in the new plan. Otherwise, after the migration, when a state is activated, the operators in the current plan, instead of the operators in the new plan, will be executed. Therefore, for an automaton state that is associated with operators in the current plan, we redirect it to be associated with the matching operators in the new plan. An operator $op$ in the current plan is
matched with another operator $op'$ in the new plan if $op'$ is a copy of $op$. In Figure 3.15, four states in the automaton, i.e., $q_2$, $q_3$, $q_7$ and $q_9$, are redirected to be associated with the operators in the new plan. For example, the association between $q_2$ and $StructuralJoin_{s_a}$ means once $q_2$ is popped off the stack, $StructuralJoin_{s_a}$ will be invoked.

Note that recording the matching relationship, i.e., remembering an operator in the new plan is copied from a certain operator in the current plan, is not an extra burden in the plan search algorithm. Even if we do not incrementally change the automaton, the plan search algorithm still has to record the matching relationship. Otherwise, after we copy the current plan and rewrite the copy, we have no way to cost the rewritten plan since the statistics are collected for the operators in the current plan.

### 3.9.2 Choosing Right Moment to Migrate

A challenge in plan migration is that the migration cannot just start at a random time, as this may corrupt the running system. The example below illustrates how corruption may arise.

**Example 15** Suppose we are running a plan in Figure 3.2. Figure 3.11 in Section 3.2.3 shows the snapshots of the stack content as the tokens are processed. Assume we now pause this plan right after we have processed a `<seller>` token and start to migrate to the new plan in Figure 3.6. The last stack in Figure 3.11 is the current stack at this moment. Since in the new plan, $b = a/seller$ is retrieved out of the automaton, the corresponding automaton of the new plan will not have states $q_4$, $q_5$ and $q_6$ as the current automaton in Figure 3.2 does. After the migration, for the
next incoming start tag, the transition entry of the state at the stack top, i.e., \( q_4 \) and 
\( q_5 \), would be looked up. However \( q_4 \) and \( q_5 \) are no longer in the automaton. This 
makes the system corrupt.

We now characterize the safe moment to start the migration. Suppose a new 
plan is derived from the current plan by mode changes of a set of pattern retrieval 
operators denoted as \( S \). We define a set \( T \) as: 
\[ T = \{ \text{Confining StructuralJoin of } navOp \mid navOp \in S \} \], where confining StructuralJoin of \( navOp \) is the 
StructuralJoin beyond which \( navOp \) cannot be moved as defined in Section 
3.6.2. We call \( T \) boundary StructuralJoins because only the subplans underneath 
these StructuralJoins are changed. We call the time that the tokens under process- 
ing are not components of any binding of \( v \) that is joined on by any boundary 
StructuralJoin (i.e., \( v \) satisfies: there exists a StructuralJoin in \( T \)) the mi-
gration window. The migration can start within the migration window.

**Example 16** In Example 15, the plan in Figure 3.6 is rewritten from the plan in 
Figure 3.2 with \( S = \{ \text{TokenNav}_{a/seller} \} \). Correspondingly, \( T = \{ \text{StructuralJoin}_{a} \} \). 
The migration can start whenever the current query plan is not in the middle of 
processing any component tokens of a binding of \( a \) (i.e., an auction element). For 
example, the migration can start right after a \(</auction> \) has been processed.

We cannot start the migration any time earlier than the migration window. Oth-
wise we can lose data. For example, suppose we start the migration in the middle 
of processing component tokens of an auction element, say, right after we have 
processed a \(</seller> \). At this moment, the output buffer of StructuralJoin in 
Figure 3.2 contains tuples each of which has two cells, one for the binding of \( b \) and 
one for the binding of \( e \). However after the migration, StructuralJoin is gone.
Note that we cannot move the tuples in the output buffer of \textit{StructuralJoin} to the output buffer of \textit{NavNest$_{b, \text{/profile}}$} in Figure 3.6, because semantically, each output tuple of \textit{NavNest$_{b, \text{/profile}}$} should contain three cells, for bindings of \$a$, \$b$ and \$e$ respectively. If we simply discard the tuples in the output buffer of \textit{StructuralJoin$_{b, \text{/profile}}$}, we then lose data.

Allowing the migration to start anytime in the migration window has impact on our migration strategy. Because the subplans that are not underneath any boundary confining \textit{StructuralJoin} operators may have tuples in their output buffers, during the migration, we must redirect these output buffers to be associated with the operators in the new plan. This redirecting process is cheap. We simply set the output buffers of these operators in the current plan to be the output buffers of the matching operators in the new plan.

Why migrating within the migration window ensures the correctness is twofold. First, no intermediate result that is not consumed yet when the migration starts will be consumed by a different set of ancestor operators after the migration compared to before the migration. Within the migration window, the query plan is not processing any bindings of \$v$ that a boundary \textit{StructuralJoin} joins on. The subplans underneath a boundary \textit{StructuralJoin} in the format of \textit{StructuralJoin$_{v}$} can generate output only when the token under processing is a component of a binding of \$v$. Since the migration window excludes the time whenever the tokens under processing are components of bindings of such \$v$, there must be no unconsumed result in the subplans underneath these boundary \textit{StructuralJoins} when the migration starts. In other words, any intermediate results unconsumed when the plan migration starts must only stay in the output buffers of those subplans which remain unchanged in the new plan. All unconsumed result generated before the
3.10. EXPERIMENTAL EVALUATION

plan migration will be consumed in the same manner as it is before the migration.

Second, suppose the mode of a $TokenNav_{v1,p,v2}$, whose confining $StructuralJoin$ is $StructuralJoin_{s_a}$, is changed. We should only remove the sub-automaton encoding the path $p$ when the states in the sub-automaton are not in the stack. These states can only be in the stack when a binding of $v1$ is being processed. A binding of $v1$ must be part of a binding of $v$. For example, in Example 16, $StructuralJoin_{s_a}$ is the confining $StructuralJoin$ of $TokenNav_{b,profile,e}$ and $TokenNav_{c,zipcode,f}$. Bindings of $b$ and $c$ are both child elements of a binding of $a$. If we pause the automaton when the element under processing is not a binding of $v$, the element under processing cannot be a binding of $v1$ either. Therefore we can safely modify the automaton without worrying about whether some states have been removed from the automaton during the migration would still remain in the stack after the migration. The situation described in Example 15 thus will not arise.

3.10 Experimental Evaluation

We have incorporated the run-time optimization techniques into the Raindrop framework. We run the experiments on two Pentium III 800 Mhz machines with 512MB memory each. One machine sends XML token streams via sockets to the second machine which would then process the received data. We count the processing time of a token from the arrival time of the token on the second machine to the time the processing on the token has been finished. The execution time of a plan on the stream is the summation of the time spent on each token in the stream.
3.10. EXPERIMENTAL EVALUATION

3.10.1 Getting Constant Values

As we have mentioned in Section 3.8, we need to get the values of the four constants \( C_{\text{nonEmp}} \), \( C_{\text{emp}} \), \( C_{\text{backtrack}} \) and \( C_{\text{bicartesian}} \). \( C_{\text{nonEmp}} \) and \( C_{\text{emp}} \) are used to evaluate the cost of a TokenNav operator (see Equation 6). \( C_{\text{visit}} \) and \( C_{\text{bicartesian}} \) are used to evaluate the cost of a NodeNav and a StructuralJoin operator respectively (see Equations 1 and 2 Section 3.2.1).

In the first experiment, we design an XML document whose root element has a tag name “root”. The root element contains \( n \) children with tag name “a”. Each element a does not have any child elements. This stream thus contains \( n + 1 \) start tokens, i.e., one <root> and \( n <a> \)'s. We also design a query “/root/a”. We construct a plan for this query which retrieves the pattern “/root/a” on the tokens. During the processing of the stream, when the <root> is encountered, the stack top must contain an initial state of the automaton. <root> matches the first navigation step “/root” and pushes a state into the stack. Next, whenever a <a> is encountered, the stack top must be non-empty. Therefore each time when a start token is encountered, the stack top is always not empty. Later, we divide the execution time spent on start tokens in the stream by \( n + 1 \) and get \( C_{\text{nonEmp}} \).

In the second experiment, we use the same XML stream and same query. However, we construct a different plan which first extracts the stream into an XML element tree and then evaluates a NodeNav operator on the tree. This NodeNav operator visits every node in the tree to retrieve the pattern “/root/a”. We divide the execution time by \( n + 1 \) and get \( C_{\text{visit}} \).

We also issue a query “/b” on the XML stream used in the first two experiments. During the processing of the stream, <root> does not match “/b” and
3.10. EXPERIMENTAL EVALUATION

Notation | Explanation | Value  
--- | --- | ---  
$C_{\text{nonEmp}}$ | average time of processing a start token when stack top is not empty | $1.361 \times 10^{-3}$ ms  
$C_{\text{emp}}$ | average time of processing a start token when stack top is empty | $0.779 \times 10^{-3}$ ms  
$C_{\text{visit}}$ | average time of visiting one node in an XML element tree | $1.622 \times 10^{-3}$ ms  
$C_{\text{bicartesian}}$ | average time of performing a binary cartesian product on one input tuple from either side to generate an output tuple | $3.012 \times 10^{-3}$ ms  

Table 3.3: Values of Constant Parameters in Cost Model

correspondingly an empty set is pushed onto the stack. Next, whenever any of the $n <a>$ tokens is encountered, the stack top is empty. We divide the execution time spent on the start tokens in the stream by $n$ and get $C_{\text{emp}}$.

To evaluate $C_{\text{bicartesian}}$, we simply run a query that involves a binary StructuralJoin operator. We divide the time spent on this StructuralJoin by the number of the cartesian product of its input tuples to get $C_{\text{bicartesian}}$. Table 3.3 gives these constant values.

3.10.2 Experiment Design for Comparing ExhaustOpt and GreedyOpt Search Strategies

Sections 3.5 and 3.6 propose an exhaustive and a greedy search algorithm, namely, ExhaustOpt and GreedyOpt. We now compare them in two aspects. First, we compare the optimization time, i.e., the time the algorithms spend on finding plans. Second, we compare the quality of the plans found by the algorithms, i.e., the execution time of the plans.

We test various queries conforming to three classes of pattern trees shown in Figure 3.16. Previous work on XQuery optimization has experimented with queries of similar structures [36, 58, 82]. In our pattern tree, a node represents an XML
Figure 3.16: Pattern Tree Templates: (a) wide and simple; (b) wide and complex; (c) deep and simple; (d) deep and complex
element. The top node in the pattern tree represents the bottom input element. The label on the edge between a parent node \(u\) and a child node \(v\) denotes an XPath \(p\), indicating there must exist descendant elements that are accessible via \(p\) from the element represented by \(u\). We now describe the characteristics of each pattern tree.

1). Figure 3.16 (a) depicts a wide pattern tree. The bottom input element in the pattern tree contains paths \(p_1, p_2, \ldots, p_n\). Each path is in the format of \(n_{11}/n_{12}/\ldots/n_{1i}[\text{filter}?]\) where \(n_{11}, n_{12}, \ldots, n_{1i}\) are element node tests and \([\text{filter}?]\) denotes an optional filter such as “/text() > 100”. We also say this tree is simple because only one node has more than one child node. In a plan that retrieves all patterns in the automaton, to retrieve an element node that has multiple child nodes, a \textit{StructuralJoin} will be performed to check whether an element contains all the required child elements. Therefore, a plan for the query in Figure 3.16 (a) contains at most one \textit{StructuralJoin}.

2). Retrieving an XML element that has more than one child in the automaton requires one \textit{StructuralJoin}. In contrast the wide pattern tree in Figure 3.16 (a) that requires only one \textit{StructuralJoin}, the wide pattern tree in Figure 3.16 (b) is more complex in the sense that it involves more \textit{StructuralJoin} operators.

3). Figure 3.16 (c) depicts a deep tree. Small linear patterns are chained together into one larger linear pattern. \textit{StructuralJoin}, which glues linear patterns into tree patterns, is not needed here. We therefore say this tree is simple.

4). In contrast to Figure 3.16 (c), Figure 3.16 (d) depicts a deep and complex pattern tree. \(n\) nodes in the tree have multiple children and thus maximally
there can be $n$ *StructuralJoins* in a Raindrop plan.

### 3.10.3 Comparing ExhaustOpt and GreedyOpt on Wide-and-Simple Pattern Trees

A pattern tree represents a class of queries. These queries locate the same patterns but return different subsets of retrieved patterns as the query results. For example, Figure 3.17 shows two query templates that both conform to the wide and simple pattern tree in Figure 3.16 (a). Query template (1) asks to return the bottom input element, i.e., $v$. All alternative plans of this query, no matter what patterns are retrieved in or out of the automaton, have to extract the same amount of data, i.e., bindings of $v$. Query template (2) asks to return $v_1$ ($v_1 = v/p_0$). Different plans can extract different amount of data. For example, a plan that retrieves $p_1$ out of the automaton still has to extract the bindings of $v$ into element nodes. In contrast, a plan that retrieves all the patterns in the automaton only needs to extract the bindings of $v_1$. For easy reference, we call Figures 3.17 (1) and (2) the *extract-same* and *extract-different* queries respectively.

When comparing the alternative plans for extract-same queries, the accuracy of costing of *Extract* operators is not important. This is because all alternative plans extract the same amount of data and thus cost the same on the *Extract* operators no matter how inaccurate *Extract* operators are costed. In contrast, the accuracy of costing of *Extract* operators is important for comparing the alternative plans for extract-different queries. In order to test the accuracy of costing of every kind of operator, we study *ExhasutOpt* and *GreedyOpt* on both extract-same and extract-different queries.
### 3.10. EXPERIMENTAL EVALUATION

```
for $v$ in $p_0[p_1][p_2]...p_n$
return $v$
```

(1)

```
for $v$ in $p_a[p_1][p_2]...p_n$
let $v1 := v/p1$
return $v1$
```

(2)

Figure 3.17: Extract-Same and Extract-Different Queries Sharing Wide and Simple Pattern Tree in Figure 3.16 (a)

### Testing Extract-Same Queries

**Query Sets:** We generate three queries that conform to the template (a.1) in Figure 3.17. These three queries differ in the number of patterns in the query, i.e., the value of $n$ in Figure 3.17 (a). The values of $n$ in the three queries are 5, 10 and 20 respectively.

**Data Sets:** We modify the DTD provided by the XML benchmark XMark [7]. We add more child elements to some elements in the XMark DTD so that we are able to issue queries that contain 20 patterns. We use ToXGene [24] to generate two XML streams conforming to the modified DTD. The size of each stream is around 52M. In XML stream 1, for any of the three queries, 4/5 of the patterns have a selectivity of 10% while 1/5 of the patterns have a selectivity of 90%. In XML stream 2, 1/5 of the patterns have a selectivity of 10% while 4/5 of the patterns have a selectivity of 90%.

The purpose of designing these two streams is to test the *ExhaustOpt* and *GreedyOpt* in the extreme cases. In XML stream 1, most pattern retrieval operators have a low selectivity. Pattern retrieval operators in the automaton are executed before those out of the automaton. The pattern retrieval operators that have low selectivities are favored to be retrieved in the automaton. Therefore, in stream...
1, the initial plan which retrieves all patterns in the automaton is close to the optimal plan. In contrast, in XML stream 2, most pattern retrieval operators have a high selectivity so that they are more favorable to be pulled out from the automaton in the initial plan. A lot of changes need to be made to the initial plan to get the final plan.

We now use ExhaustOpt and GreedyOpt to generate plans for the three queries on both streams 1 and 2. We run an initial plan that retrieves all patterns in the automaton on the stream, collect statistics from the stream and apply the search algorithm to get a new plan. We then run the new plan on the same stream again and measure its execution time. Table 3.4 reports the result.

The patterns \( p_1, \ldots, p_n \) in Figure 3.17 (a.1) are all siblings. Therefore any combinations among \( p_1, \ldots, p_n \) are valid (combinations of ancestor-descendant patterns are invalid according to Lemma 2 in Section 3.5). The number of alternative plans explored in ExhaustOpt is then \( 2^n \). We can see that when \( n = 10 \), the optimization time already far exceeds the execution time on both XML streams 1 and 2 (Rows 2 and 5 in Table 3.4). When \( n = 20 \), the number of alternative plans explored by ExhaustOpt explodes and makes ExhaustOpt obviously impractical. Hence we do not report it here.

The number of plans explored by ExhaustOpt is fixed given a query. That is why ExhaustOpt explores 32 and 1024 plans on both XML streams 1 and 2 when \( n = 5 \) and 10 respectively. In contrast, the number of plans explored by GreedyOpt can vary with different streams. For the same query, GreedyOpt on XML stream 1 explores less plans than on XML stream 2. This is because GreedyOpt terminates when no mode change of a pattern retrieval in the current plan yields a better plan.

Although GreedyOpt explores much less plans than ExhaustOpt, it generates
<table>
<thead>
<tr>
<th>Stream 1</th>
<th>Stream 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
</tr>
<tr>
<td>10</td>
<td>∞</td>
</tr>
<tr>
<td>20</td>
<td>∞</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
</tr>
<tr>
<td>20</td>
<td>∞</td>
</tr>
</tbody>
</table>

Table 3.4: *ExhaustOpt* and *GreedyOpt* for Extra-Same Queries in Figure 3.17 (1)
the same plan as *ExhaustOpt* (compare the columns of “Plan Exec. Time” in *ExhaustOpt* with that in *GreedyOpt*. *GreedyOpt* succeeds to final optimal plans in all cases in this experiment setting.

The last two columns in Table 3.4 summary the “effectiveness” of both *ExhaustOpt* and *GreedyOpt*. We define “effectiveness” of a search strategy as (the time spent on finding a plan + the time spent on executing the plan found)/(the time spent on executing the initial plan). The less the number is (i.e., spent less time on finding a plan that runs faster), the more effective the search algorithm is. *GreedyOpt* is more effective in stream 2 than in stream 1. This is because in stream 1, the initial plan is close to the optimal plan while in stream 2, the initial plan is significantly worse than the optimal plan.

**Testing Extract-Different Queries**

We now evaluate the extract-different queries conforming to the template (2) in Figure 3.17 on the two XML streams. Alternative plans of an extract-different query extract different amount of data. Table 3.5 shows the result. Again, for the three queries on both streams, *GreedyOpt* finds the same plan as *ExhaustOpt* but in much less time than *ExhaustOpt*. In Stream 1, the initial plan itself is the optimal plan. The plan search is a pure overhead. However, when $n = 10$ or 20, the overhead is ignorable, taking 4% or 3% of the overall execution time respectively. In stream 2, the plan found by *GreedyOpt* cuts down the execution time of the initial plan by 20% to 40%.
### Table 3.5: *ExhaustOpt* and *GreedyOpt* on Extract-Different Queries in Figure 3.17 (a.2)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream 1</td>
<td>5</td>
<td>32</td>
<td>502</td>
<td>502</td>
<td>1248</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1024</td>
<td>15306</td>
<td>5042</td>
<td>15306</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>∞</td>
<td>N/A</td>
<td>20</td>
<td>302</td>
</tr>
<tr>
<td>Stream 2</td>
<td>5</td>
<td>32</td>
<td>516</td>
<td>3902</td>
<td>3902</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1024</td>
<td>14120</td>
<td>9123</td>
<td>14120</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>∞</td>
<td>N/A</td>
<td>204</td>
<td>3104</td>
</tr>
</tbody>
</table>

*Table 3.5: ExhaustOpt and GreedyOpt on Extract-Different Queries in Figure 3.17 (a.2)*
3.10. EXPERIMENTAL EVALUATION

3.10.4 Comparing ExhaustOpt and GreedyOpt on Wide-and-Complex Pattern Trees

We now compare the ExhaustOpt and GreedyOpt for wide-and-complex queries conforming to the template in Figure 3.16 (b). Our experiments consist of two parts. In the first part, we test on a set of data streams with varying data characteristics. The purpose is to observe how GreedyOpt behaves on relatively “random” data sets. In the second part, we focus on studying when GreedyOpt fails to find the optimal plans.

We generate XML streams conforming to the DTD describing Ebay’s auction data from University of Washington’s XML repository [60]. The root element contains a sequence of listing child elements. The DTD of a listing element is as follows: 

```xml
< !ELEMENT listing (seller_info, payment_types, shipping_info, buyer_protection_info, auction_info, bid_history, item_info) >
```

Among the seven child elements of listing, four of them (e.g., seller_info and auction_info) have nested structures, i.e., they can have children again. We design a query, shown in Figure 3.18, which navigates into each nested element. For each nested element, we pose a filter on each of its child elements. More specifically, \( b \), \( c \), \( d \) and \( e \) have 2, 2, 12 and 5 child elements and thus 2, 2, 12 and 5 filters respectively.

Table 3.6 shows the data characteristics of four XML streams conforming to the DTD. “Sel.” in the table denotes the abbreviation we use for selectivity.

The query in Figure 3.18 contains 25 patterns whose modes can be changed (i.e., \( b \), \( c \), \( d \), \( e \) and their 21 filters). The number of alternatives that will be explored by ExhaustOpt is so large that ExhaustOpt is clearly impractical. Therefore we terminate ExhaustOpt after it has explored 1000 plans and return the best plan.
for $a$ in /listing
let $b := a/seller\_info[seller\_rating > 4][seller\_name contains "SF"];
$e := a/bid\_history[...][...];$
$d := a/auction\_info[...][...];$
$e := a/item\_info[...][...]
where $b$ and $c$ and $d$ and $e$
return $a$

Figure 3.18: Wide-and-Complex Query on Ebay Data: requiring to return a listing whose $a/seller\_info$, $a/bid\_history$, $a/auction\_info$, and $a/item\_info$ satisfy 2, 2, 12, and 5 Filters Respectively

<table>
<thead>
<tr>
<th>Stream</th>
<th>Sel. of $b$</th>
<th>Sel. of $c$</th>
<th>Sel. of $d$</th>
<th>Sel. of $e$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10%</td>
<td>50%</td>
<td>70%</td>
<td>90%</td>
</tr>
<tr>
<td>2</td>
<td>90%</td>
<td>10%</td>
<td>50%</td>
<td>70%</td>
</tr>
<tr>
<td>3</td>
<td>70%</td>
<td>90%</td>
<td>10%</td>
<td>50%</td>
</tr>
<tr>
<td>4</td>
<td>50%</td>
<td>70%</td>
<td>90%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Table 3.6: Random Data Sets Conforming to Ebay’s DTD: Each Stream around Size 55M

among these 1000 plans. Table 3.7 compares ExhaustOpt and GreedyOpt for the query in Figure 3.18 on the streams in Table 3.6.

<table>
<thead>
<tr>
<th>Stream</th>
<th>ExhaustOpt</th>
<th>GreedyOpt</th>
<th>Initial Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream 1</td>
<td>30072</td>
<td>15043</td>
<td>57</td>
</tr>
<tr>
<td>Stream 2</td>
<td>25087</td>
<td>14893</td>
<td>59</td>
</tr>
<tr>
<td>Stream 3</td>
<td>24508</td>
<td>16012</td>
<td>76</td>
</tr>
<tr>
<td>Stream 4</td>
<td>42301</td>
<td>15567</td>
<td>37</td>
</tr>
</tbody>
</table>

Table 3.7: ExhaustOpt and GreedyOpt for Query in Figure 3.18 on XML Streams in Figure 3.6 (ExhaustOpt Limited to Explore 1000 Plans)

In Streams 1 and 4, ExhaustOpt fails to find any plan better than the initial plan in the first 1000 plans it has explored. This is because selectivity of $b$ is rather low so that the optimal plan retrieves $b$ in the automaton. When we call
ExhaustOpt, we pass it a parameter navsToBeTried (see Algorithm 7), which is a list of pattern retrieval operators whose modes would be changed. The first operator appearing in the navsToBeTried list happens to be $b$. ExhaustOpt thus explores all alternative plans with $b$ retrieved out of the automaton first. These plans are all worse than the initial plan so that EnumSearch explores the first 1000 alternative plans to no avail. GreedyOpt instead makes steady progress to finding a better plan during each iteration. On all four streams, GreedyOpt explores a limited number of alternative plans yet in all cases it finds a plan that cuts the initial execution time by 15% to 56%.

3.10.5 Comparing ExhaustOpt and GreedyOpt on Deep-and-Simple Pattern Trees

We now compare the ExhaustOpt and GreedyOpt for deep-and-simple queries conforming to the template in Figure 3.19. According to a DTD survey [16], the depth of an XML document is usually less than 8. Therefore we limit the number $n$ in Figure 3.20 to be less than 8. We generate a XML stream in which all patterns in the queries have the same selectivity of 70%. Table 3.8 compares ExhaustOpt and GreedyOpt for the queries in Figure 3.19 on this stream.

```xml
for $v$ in $p_0$, $v_1$ in $v/p_1$, $v_2$ in $v_1/p_2$, ..., $v_n$ in $v_{n-1}/p_n$
return <result> $v$, $v_1$, ..., $v_n</result>
```

Figure 3.19: Queries Conforming to Wide-and-Deep Pattern Tree in Figure 3.16 (c)

We observe two phenomena in Table 3.8 as follow.
3.10. EXPERIMENTAL EVALUATION

Table 3.8: ExhaustOpt and GreedyOpt for Deep-and-Simple Pattern Trees on XML Stream with Size of 51M

<table>
<thead>
<tr>
<th>n</th>
<th>ExhaustOpt</th>
<th>GreedyOpt</th>
<th>Initial Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>125</td>
<td>3790</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>123</td>
<td>3892</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>150</td>
<td>4012</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>145</td>
<td>3991</td>
</tr>
</tbody>
</table>

1). *ExhaustOpt* and *GreedyOpt* explore exactly the same set of alternative plans. This is because every pair of pattern retrieval in the plan has pattern dependency relationship. As long as one pattern retrieval has been moved out in the initial plan, no other pattern retrieval can be further moved out in the newly derived plan. Therefore after exploring *n* alternative plans each of which corresponds to moving out one pattern retrieval in the initial plan, both *ExhaustOpt* and *GreedyOpt* terminate.

2). No matter what the value of *n* is, the best plan is always the one which retrieves all patterns in the automaton. Due to the pattern dependency, $p_{31}$ must be retrieved after $p_{21}$; $p_{41}$ must be retrieved after $p_{31}$ and so on. Regardless of whether these patterns are retrieved in or out, the execution order is always serialized. Retrieving these patterns out of the automaton does not provide any extra benefit. The plan in which all pattern retrieval is pushed into the automaton ensures that the least amount of data is buffered.
3.10. EXPERIMENTAL EVALUATION

3.10.6 Comparing ExhaustOpt and GreedyOpt on Deep-and-Complex Pattern Trees

It is interesting to observe that for queries conforming to the deep-and-complex pattern tree in Figure 3.16 (d), GreedyOpt terminates very quickly. According to Lemma 2 in Section 3.5, two operators that have pattern dependency relationship cannot both undergo mode changes. Suppose from a current plan, the mode change on $p_{i2}$ ($1 < i < n$) in Figure 3.16 is chosen, then the mode changes on its ancestor and descendant patterns, including $p_{11}$, $p_{21}$, ..., $p_{(i-1)1}$, will no longer be considered. Suppose the mode change on $p_{i1}$ is chosen, then even more mode changes are disqualified for consideration, including mode changes on patterns $p_{11}$, $p_{21}$, ..., and $p_{n1}$.

To illustrate the property of quick termination of GreedyOpt, we test the queries conforming to the template in Figure 3.20. We then run these queries on the same XML stream used in Section 3.10.5. Table 3.9 reports the results.

```xml
for $v$ in $p_0$
  let $v_{11} := v/p_{11}$,
  $v_{12} := v/p_{12}$,
  $v_{21} := v_1/p_{21}$,
  $v_{22} := v_1/p_{22}$,
  ..., 
  $v_{n1} := v_{n-1}/p_{n1}$,
  $v_{n2} := v_{n-1}/p_{n2}$
where $v_{11}$ and $v_{12}$ and ... $v_{n1}$ and $v_{n2}$
return $v$
```

Figure 3.20: Queries Conforming to Wide-and-Complex Pattern Tree in Figure 3.16 (d)
### 3.10. Experimental Evaluation

<table>
<thead>
<tr>
<th>n</th>
<th>ExhaustOpt</th>
<th>GreedyOpt</th>
<th>Initial Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>147</td>
<td>2296</td>
<td>7356</td>
</tr>
<tr>
<td>4</td>
<td>595</td>
<td>8674</td>
<td>10086</td>
</tr>
<tr>
<td>5</td>
<td>2387</td>
<td>38500</td>
<td>12176</td>
</tr>
<tr>
<td>6</td>
<td>9555</td>
<td>180078</td>
<td>13408</td>
</tr>
</tbody>
</table>

Table 3.9: ExhaustOpt and GreedyOpt for Deep-and-Complex Ternary Pattern Trees on XML Stream with Size of 51M

Even for the queries involving a large number of patterns, GreedyOpt terminates rather quickly. Let us use the last row in Table 3.9 as an example. When \( n = 6 \), i.e., the depth of the pattern tree in Figure 3.16 (c) is 6. There are 18 patterns in total in the tree. ExhaustOpt explores 9555 alternatives while GreedyOpt only explores 20 alternatives. Even though GreedyOpt fails to find the optimal plan, the plan it finds still cuts down the execution time of the initial plan by 29%.

#### 3.10.7 Study on when GreedyOpt Fails to Find Optimal Plan

We now investigate when GreedyOpt may fail to find the optimal plans. We study extract-same and extract-different queries, shown in Figure 3.21, conforming to the wide-and-complex pattern tree in Figure 3.16 (b) with \( n = 5 \). Since each \( p_i \) (\( 1 < i < n \)) has two child patterns \( p_{i1} \) and \( p_{i2} \), there are 15 patterns in the query in total.

For each query, we perform extensive experiments on different data sets. We also test with different initial plans. Note that in the one-time optimization scenario, we always use an initial plan that retrieves all patterns in the automaton. However, in the continuous optimization scenario, the initial plan of each optimization is the plan found in the last optimization. Therefore the initial plan can be any kind of plans. We find that in two cases GreedyOpt fails to find the optimal
for $v$ in $p_0$,
where $v/p_1[p_{11}] [p_{12}]$
and ...
and $v/p_2[p_{21}] [p_{22}]$
return $v$

(1)

for $v$ in $p_0$,
let $v_1 := v/p_1$
where $v/p_1[p_{11}] [p_{12}]$
and ...
and $v/p_2[p_{21}] [p_{22}]$
return $v_1$

(2)

Figure 3.21: Extract-Same and Extract-Different Queries Conforming to Wide-
and-Complex Pattern Tree in Figure 3.16 (b)

plans.

Case 1: Missing Synergy Benefits

In the first case, GreedyOpt fails to find the optimal plan of the extract-different
query in Figure 3.21 (b). The characteristics of this case are as below. The initial
plan retrieves all the patterns but $p_{11}$ and $p_{12}$ in the automaton. The optimal plan
found by ExhaustOpt retrieves all patterns in the automaton. GreedyOpt fails to
find the optimal plan. In the first iteration, GreedyOpt changes the mode of one
pattern retrieval at a time. No single mode change leads to a better plan in this
iteration. GreedyOpt then terminates.

However ExhaustOpt finds that if it pushes both $p_{11}$ and $p_{12}$ into the automa-
ton, $v$ does not need to be extracted. Instead, only $v_1$ needs to be extracted. This
way we cut the extraction cost by (cost of extracting $v$ - cost of extracting $v_1$).
If the cost cut is large enough, then pushing in $p_{11}$ and $p_{12}$ can yield a better plan
than the initial plan. However this better plan is not considered by GreedyOpt.
GreedyOpt only considers a mode change on one single pattern retrieval at each
time. When all single mode changes fail, GreedyOpt would not further explore the synergy that may result from the combination of two mode changes.

To experimentally illustrate this case, we design two XML streams as below.

1). In XML stream 1, children of bindings of $v_1$ in Figure 3.16 (b) are bound to either $v_{11}$ or $v_{12}$. Therefore extracting the bindings of $v_1$ costs almost the same as extracting the bindings of $v_{11}$ and $v_{12}$.

2). In XML stream 2, bindings of $v_1$ contain many children other than bindings of $v_{11}$ and $v_{12}$. Therefore extracting bindings of $v_1$ costs significantly more than extracting only bindings of $v_{11}$ and $v_{12}$.

For each stream, we design two queries as below.

1). In query 1, $p_{11}$ and $p_{12}$ have low costs and low selectivity. There is also a costly filter in the format of “$v_{n1}/text()$ contains ...”. Therefore $p_{11}$ and $p_{12}$ are favored to be retrieved in the automaton. Doing so reduces the cost of the costly filter.

2). In query 2, $p_{11}$ and $p_{12}$ have high costs and high selectivity. All the other patterns however have low costs and low selectivity. Therefore $p_{11}$ and $p_{12}$ are favored to be retrieved out of the automaton.

Combining the above two XML streams and two queries, we get the four settings in Figure 3.10. For each setting, we test both the extract-same and extract-different queries in Figure 3.21. Therefore there are eight settings in total. For each setting, we apply ExhaustOpt and GreedyOpt on an initial plan that retrieves all patterns but $p_{11}$ and $p_{12}$ in the automaton. The results are reported in Figure 3.22.
Table 3.10: Environment Settings for Testing Case of “Missing Synergy Benefits”

We see that GreedyOpt works well in most of these 8 cases. It only fails to find the optimal plan in the setting 4 (see the highlighted row in Figure 3.22). Note that GreedyOpt for the extract-same query with the same setting (the row in italics font) however yields the optimal result. The “extra synergy benefits” save the extraction cost of $v_1$ when all children of $v_1$ are retrieved in the automaton. However if $v_1$ has to be extracted anyway, then this cost cannot be saved no matter whether the patterns are retrieved in or out of the automaton.

**Case 2: Accounting of Cost Cut from Secondary Effect**

In the second case, GreedyOpt fails to find the optimal plans for both queries in Figure 3.21. The characteristics of this case are as below. In the first iteration, GreedyOpt finds that pulling out $p_{11}$ alone and pulling out $p_{12}$ alone generates two better plans respectively. However pulling out $p_1$ also causes $p_{11}$ and $p_{12}$ to be pulled out. This is called secondary effect when a pattern with descendant patterns is pulled out (see Section 2.4.3). Pulling out $p_1$ can yield a plan that is even better than the two previous ones. GreedyOpt then chooses to pull out $p_1$. This new
### Table 1: Comparison of ExhaustOpt and GreedyOpt for Environment Settings

<table>
<thead>
<tr>
<th>Setting Used</th>
<th>ExhaustOpt</th>
<th>GreedyOpt</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of plans explored</td>
<td>Plan Chosen</td>
</tr>
<tr>
<td>Buffer-Same</td>
<td>1 3124</td>
<td>no change on initial plan</td>
</tr>
<tr>
<td></td>
<td>2 3124</td>
<td>$p_{11}$ &amp; $p_{12}$ pushed in</td>
</tr>
<tr>
<td></td>
<td>3 3124</td>
<td>no change on initial plan</td>
</tr>
<tr>
<td></td>
<td>4 3124</td>
<td>$p_{11}$ &amp; $p_{12}$ pushed in</td>
</tr>
<tr>
<td>Buffer-Different</td>
<td>1 3124</td>
<td>no change on initial plan</td>
</tr>
<tr>
<td></td>
<td>2 3124</td>
<td>$p_{11}$ &amp; $p_{12}$ pushed in</td>
</tr>
<tr>
<td></td>
<td>3 3124</td>
<td>no change on initial plan</td>
</tr>
<tr>
<td></td>
<td>4 3124</td>
<td>$p_{11}$ &amp; $p_{12}$ pushed in</td>
</tr>
</tbody>
</table>

Figure 3.22: ExhaustOpt and GreedyOpt for Environment Settings in Figure 3.10 Illustrating “Missing Synergy Benefits”. Initial Plan Used: All Patterns but $p_{11}$ and $p_{12}$ Retrieved in Automaton.
3.10. EXPERIMENTAL EVALUATION

plan can actually lose to a plan resulted from pulling out both $p_{11}$ and $p_{12}$ but not $p_1$. The cost cut of pulling out $p_1$ may come from its secondary effect. In short, *GreedyOpt* accounts the cut cost to a mode change while the credits should actually be given to the secondary effect.

We design three settings in Figure 3.11. In all settings, $p_{11}$ and $p_{12}$ are inclined to be retrieved out of the automaton because of their high selectivities and high costs.

<table>
<thead>
<tr>
<th>setting</th>
<th>Data Characteristics</th>
<th>Query Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size of $v$/size of $v_1$</td>
<td>selectivity of $p_1$</td>
</tr>
<tr>
<td>1</td>
<td>100%</td>
<td>50%</td>
</tr>
<tr>
<td>2</td>
<td>150%</td>
<td>50%</td>
</tr>
<tr>
<td>3</td>
<td>150%</td>
<td>90%</td>
</tr>
</tbody>
</table>

Table 3.11: Environment Settings for Testing Case of “Wrong Accounting of Cost Cut”: Size of XML Stream 1, 2 and 3 is 42, 62, 59M Respectively

XML streams 1 and 2 differ in the ratio of the size of bindings of $v$ to the size of bindings of $v_1$. The first two rows in Figure 3.23 show the results of applying ExhaustOpt and GreedyOpt for the extract-different query. The initial plan retrieves all patterns in the automaton. For XML stream 1, the difference between the non-optimal plan chosen by GreedyOpt and the optimal plan chosen by ExhaustOpt is not significant because the cost of buffering the bindings of $v$ is close to that of buffering the bindings of $v_1$. In contrast, for XML stream 2, the cost difference of the two plans is more significant due to the increased difference in their buffering costs.

The XML streams 2 and 3 differ in the selectivity of $v$. The last two rows in Figure 3.23 show the results of applying *ExhaustOpt* and *GreedyOpt* for the
extract-same query. The initial plan retrieves all patterns in the automaton. In XML stream 3, selectivity of $p_1$ is rather low. Pulling out $p_1$ is thus not chosen. Therefore $GreedyOpt$ still finds the optimal plan. In XML stream 2, selectivity of $p_2$ is higher than in XML stream 1. This time pulling out $p_1$ is chosen while actually pulling out $p_{11}$ and $p_{12}$ only is even better. $GreedyOpt$ fails to find optimal plan.

In summary, $GreedyOpt$ on both buffer-different and buffer-same queries can fail to find the optimal plans because of a wrong accounting of the cost cut.

<table>
<thead>
<tr>
<th>Setting Used</th>
<th>ExhaustOpt</th>
<th>GreedyOpt</th>
<th>Initial Plan Exec. Time (ms.)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of plans explored</td>
<td>Plan Chosen</td>
<td>Plan Exec. Time (ms.)</td>
</tr>
<tr>
<td>Buffer-Different Query</td>
<td>3124</td>
<td>$p_{11}$ &amp; $p_{12}$ pulled out</td>
<td>15502</td>
</tr>
<tr>
<td>Buffer-Same Query</td>
<td>3124</td>
<td>$p_{11}$ &amp; $p_{12}$ pulled out</td>
<td>18045</td>
</tr>
<tr>
<td>Buffer-Same Query</td>
<td>3124</td>
<td>$p_{11}$ &amp; $p_{12}$ pulled out</td>
<td>18507</td>
</tr>
</tbody>
</table>

Figure 3.23: $ExhaustOpt$ and $GreedyOpt$ Comparison for Settings in Figure 3.10 illustrating “Wrong Accounting of Cost Cut”. Initial Plan Used: All Patterns Retrieved in Automaton.

**Conclusion**

The case study in Section 3.10.7 sheds some lights on how we can further improve $GreedyOpt$. In case 1, in order not to missing synergy benefits, we can improve the termination criterion in the $GreedyOpt$ algorithm. Currently, $GreedyOpt$ terminates when no single mode change leads to a better plan than the current plan. Instead, we can further check whether multiple mode changes can lead to a bet-
ter plan. In case 2, in order to correctly account the cost cut, we can improve the
criterion of which plan to adopt as the current plan in the GreedyOpt algorithm.
Suppose a best plan in a search iteration results from a mode change that has sec-
ondary effects. Currently, we adopt this best plan, denoted as P, as the current plan.
Instead, we can also cost a plan $P'$ resulted from only the secondary mode changes.
If $P'$ is better than $P$, we then adopt $P'$ as the current plan.

3.10.8 Comparison of GreedyOpt and GreedyPruneOpt

In continuous optimization scenarios, we now drop the optimization dimension of
reordering input subplans to StructuralJoin. Correspondingly, greedy algorithm
can now be made more efficiently with pruning rules. In the previous sections, the
size of XML streams we use various between 40M - 60M. To study the continuous
optimization scenario, we now assume the statistics change for every 20M - 30M
of XML stream. We then compare Greedy and Greedy with pruning on an XML
document about 20M - 30M. We repeat the same queries with the XML streams of
the same data characteristics as in Figure ???. The only difference is that the size of
the XML stream is now 25M instead of 52M used in Figure ???

<table>
<thead>
<tr>
<th>Setting</th>
<th>$n$</th>
<th>Greedy</th>
<th>Greedy with Pruning</th>
<th>Initial Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream 1</td>
<td>5</td>
<td>9</td>
<td>232</td>
<td>782</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>27</td>
<td>475</td>
<td>2865</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>144</td>
<td>2721</td>
<td>4821</td>
</tr>
<tr>
<td>Stream 2</td>
<td>5</td>
<td>15</td>
<td>381</td>
<td>2032</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>54</td>
<td>823</td>
<td>4742</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>204</td>
<td>3126</td>
<td>11405</td>
</tr>
</tbody>
</table>

Figure 3.24: Greedy and Greedy with Pruning for Buffer-Same Queries in Figure 3.17 (a.1)
3.10. EXPERIMENTAL EVALUATION

From Figure 3.24 we can see that Greedy with pruning cuts down the number of plans explored in all six experiments. For wide and simple queries, no patterns have descendant patterns. Therefore the technique of “pruning by bounding cost cut” described in Section 3.7 takes effect. This technique excludes the pull-out of those $TokenNav$ with selectivity of 10%. Moreover, it also cuts down the unit time spent on processing each alternative plan since we no longer apply the optimization using the input-subplan-reordering rule. Among all six experiments, the pruning technique improves the optimization time most significantly for row 3 in Figure 3.24 since the initial plan has more $TokenNav$ operators that have a selectivity of 10% than any of the other five initial plans.

3.10.9 Overhead of One-time Optimization: From Statistics Collection to Plan Migration

The overhead of run-time optimization is composed of three components, i.e., statistics collection, plan search and plan migration (if any). We study the overhead of each of the three components in the one-time optimization scenario. Since we have already studied the overhead of the plan search time when comparing ExhaustOpt and GreedyOpt in Section 3.10.2, we now focus on the overhead of statistics collection and plan migration.

**Query Sets:** We design two queries both of which conform to the template in Figure 3.21 (a) but differ in the number of patterns in the query ($n$ in Figure 3.21 is 5 and 10 respectively; each node has exactly two children). We can compare the overhead of statistics collection in the execution of these two queries, since a query involving more patterns spends more time in statistics collection.

**Data Sets:** We also design two streams (the design principle is similar to that
3.10. EXPERIMENTAL EVALUATION

for XML streams used in Section 3.10.3. In XML stream 1, for either queries mentioned above, 4/5 of $p_1$, ..., and $p_n$ have a selectivity of 10% while the rest 1/5 have a selectivity of 90%. In XML stream 2, only 1/5 of $p_1$, ..., and $p_n$ have a selectivity of 10% while the rest 4/5 have a selectivity of 90%. In both streams, all child patterns of $p_1$, ..., and $p_n$ have the same selectivity as their parent patterns. For a query runs on XML stream 1, the optimal plan is only slightly different from the initial plan which retrieves all patterns in the automaton. In contrast, when the same query is run on XML stream 2, its optimal plan undergoes more dramatic changes from the initial plan. We therefore can compare the overhead of a simple plan migration with a more complicated plan migration process.

Given the above two queries and two queries, we have four experiment settings. Figure 3.25 reports the result in the four experiments. For each query, we illustrate the four cost ingredients of query processing with run-time optimization, i.e., (1) the plan execution time, i.e., the execution time of initial plan + the execution time of the optimized plan, (2) the plan search time by GreedyOpt algorithm, (3) the time for statistics collection and (4) the plan migration time. The costs of the latter three is the overhead of the run-time optimization. We can see that in all four experiments, the plan search time dominates the overhead. The time of statistics collection ranges from 10ms - 20ms while that of plan migration ranges from 0ms - 40ms (the statistics collection time and the plan search time are so small that they are almost unrecognizable in Figure 3.25). Table 3.12 further compares the query processing time without the run-time optimization with that with the run-time optimization. In all four experiments, the query processing with run-time optimization has better performance than that without run-time optimization.
3.10. EXPERIMENTAL EVALUATION

Run-time Plan Optimization Overhead (One-time Optimization Scenario)

Figure 3.25: Cost Ingredients of Query Processing in One-time Optimization

<table>
<thead>
<tr>
<th>Setting</th>
<th>Query Processing Time without Run-time Optimization (ms)</th>
<th>Query Processing Time with Run-time Optimization (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8690</td>
<td>7982</td>
</tr>
<tr>
<td>2</td>
<td>18828</td>
<td>16506</td>
</tr>
<tr>
<td>3</td>
<td>17331</td>
<td>9635</td>
</tr>
<tr>
<td>4</td>
<td>28415</td>
<td>18135</td>
</tr>
</tbody>
</table>

Table 3.12: Comparison of Query Processing Time with and without Run-time Optimization
3.10.10 Performance of Continuous Optimization

We have studied the plan search performance in the continuous optimization scenario by comparing Greedy and Greedy with pruning in Section 3.10.8. We have also shown in Section 3.10.9 that both statistics collection and plan migration are very cheap. In this section, we focus on the effect of continuous optimization on the query processing rate, i.e., number of bottom input elements processed per second.

We use the buffer-same query that is also used for experiments in Figure 3.22. We generate four XML fragments each of which contains 2500 auctions. The data characteristics of these XML fragments are shown in Figure 3.10. We concatenate these four XML fragments into one stream. If we denote a plan that retrieves all the patterns in the automaton as $P_1$, and a plan that pulls out $p_{11}$ and $p_{12}$ as $P_2$. According to Figure 3.22 (first four rows), the run-time optimization will lead to the following plan changes ($P_1 \rightarrow P_2$ denotes $P_1$ is changed to $P_2$): $P_1 \rightarrow P_2 \rightarrow P_1 \rightarrow P_2 \rightarrow P_1$. We start optimization every 500 auctions.

We compare the two plan execution processes, one with the run-time optimization and one without run-time optimization. Figure 3.26 shows the processing rate over time. For plan execution without run-optimization, there are four periods in each of which the processing rate is rather consistent. For plan execution with run-time optimization, there are two small time windows (around 18s and 28s) in which the processing rates are significantly lower than those in its neighboring time windows. These two windows indicate the time when optimization for XML stream fragments 3 and 4 happens. Since the query engine spends time (0.2s and 0.6s respectively) on plan search without processing any input, the processing rates decrease. The optimization for XML fragment 1 happens around the 3rd second.
so that we can see the processing rate starts to increase from this point. The optimization for XML fragment 2 happens around 8s. There is however not an obvious processing rate decrease as that for XML fragments 3 and 4. This is because the plan chosen for XML fragment 2 is faster than any plans chosen for other XML fragments. The plan search for XML fragment 2 takes 0.6s, but in the rest of 1.6s the processing rate is rather high. So on average the processing rate is not significantly lower than before.

![Graph showing processing rate comparison with and without runtime optimization](image.png)

Figure 3.26: Processing Rate of Wide and Complex Query in Continuous Optimization Scenario