

# Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries

Jianjun Chen David J. DeWitt Jeffrey F. Naughton  
Computer Sciences Department  
University of Wisconsin-Madison  
{jchen, dewitt, naughton}@cs.wisc.edu

## Abstract

*In this paper, we design and evaluate alternative selection placement strategies for optimizing a very large number of continuous queries in an Internet environment. Two grouping strategies, PushDown and PullUp, in which selections are either pushed below, or pulled above, joins are proposed and investigated. While our earlier research has demonstrated that the incremental group optimization can significantly outperform an ungrouped approach, the results from this paper show that different incremental group optimization strategies can have significantly different performance characteristics. Surprisingly, in our studies, PullUp, in which selections are pulled above joins, is often better and achieves an average 10-fold performance improvement over PushDown (occasionally 100 times faster). Furthermore, a revised algorithm of PullUp, termed filtered PullUp is proposed that is able to further reduce the cost of PullUp by 75% when the union of the selection predicates is selective. Detailed cost models, which consider several special parameters, including (1) characteristics of queries to be grouped, and (2) characteristics of data changes, are presented in this paper. Preliminary experiments using an implementation of both strategies show that our models are fairly accurate in predicting the results obtained from the implementation of these techniques in the Niagara system. This work can serve as the basis for building a cost-based incremental group query optimizer to choose a better grouping strategy.*

## 1. Introduction

Continuous queries [CDTW00][TGNO92][LPT99] are persistent queries that allow users to receive new results when they become available. While continuous query systems can transform a passive web into an active environment, such systems must be able to support a large number of queries due to the scale of the Internet. In [CDTW00], we addressed this problem by grouping continuous queries based on the observation that many

web queries share similar structures. Grouped queries can share the common computation, tend to fit in memory, and can reduce I/O costs significantly. Furthermore, grouping on selection predicates can eliminate a large number of unnecessary query invocations.

Our grouping technique is distinguished from previous multiple query optimization approaches [CM86] [RC88] [Sel86] [RSSB00] in that we use an incremental grouping strategy. These earlier strategies focused on finding an optimal global plan for a small number of queries submitted in advance. A naive approach for grouping continuous queries would be to apply these methods directly by re-optimizing all queries whenever a new query is added. We contend that such an approach is not acceptable for large, dynamic environments, in which queries are continuously added and removed, because of the associated performance overhead. In our approach, when a new query is submitted, the group optimizer considers existing groups as potential optimization choices by using either cost-based heuristics or a slightly-modified cost-based query optimizer. The new query is merged into those existing groups that match its signatures. Existing queries are not, however, re-grouped. Preliminary results [CDTW00] demonstrate that incremental group optimization can significantly improve the execution time when compared to an approach without grouping. The experimental results also show that the approach scales to support a large number of queries. Other important advantages and some limitations of our approach are detailed in our previous paper [CDTW00].

[CDTW00] proposed two strategies, pushing down selections below joins or pulling up selections above joins (termed PushDown or PullUp in this paper), for choosing the order for grouping selection and join operators, but only evaluated the performance of the PushDown approach. In this paper, we investigate the two strategies in detail and propose a revised version of the PullUp algorithm. Our results show that different incremental group optimization strategies can exhibit significantly different performance characteristics under different conditions. Surprisingly, in our studies, PullUp is often

better and achieves an average 10-fold performance improvement over PushDown.

In this paper we propose a cost model for incremental group optimization. We consider several special parameters including (1) characteristics of the queries to be grouped, e.g. the number of queries and the distribution of distinct constant values; and (2) characteristics of the data changes, e.g. the update frequency and update distribution. Since intermediate query results are materialized in our approach, the cost of maintaining materialized views is also included in the cost models. This work can serve as the basis for building a cost-based incremental group query optimizer to choose a better grouping strategy.

Section 2 presents a brief review of the main ideas of incremental group optimization techniques described in [CDTW00]. Section 3 discusses the PushDown and PullUp strategies. An analytic analysis of these two strategies is given in Section 4. Section 5 examines the performance of the two strategies. Section 6 discusses related work and Section 7 concludes the paper.

## 2. Review of Incremental Group Optimization

To make this paper self-contained, in this section, we give a brief review of the main ideas of incremental group optimization [CDTW00]. For illustration purposes, we use the two XML-QL queries in Figure 2.1 to illustrate the main ideas of our incremental group optimization techniques. The queries in Figure 2.1 retrieve stock quotes

<pre>Where &lt;Quotes&gt;&lt;Quote&gt;&lt;Symbol&gt;\$s&lt;/&gt; &lt;Price&gt;\$p&lt;/&gt;&lt;/&gt;   element_as \$g &lt;/&gt; in "quotes.xml",    \$p &gt; 90 &lt;Companies&gt;&lt;Company&gt;&lt;Symbol&gt;\$s&lt;/&gt;&lt;/&gt;   element_as \$t &lt;/&gt; in "profiles.xml"   construct \$g, \$t</pre>	<pre>Where &lt;Quotes&gt;&lt;Quote&gt;&lt;Symbol&gt;\$s&lt;/&gt; &lt;Price&gt;\$p&lt;/&gt;&lt;/&gt;   element_as \$g &lt;/&gt; in "quotes.xml",    \$p &gt; 100 &lt;Companies&gt;&lt;Company&gt;&lt;Symbol&gt;\$s&lt;/&gt;&lt;/&gt;   element_as \$t &lt;/&gt; in "profiles.xml"   construct \$g, \$t</pre>
--	---

Figure 2.1: Two XML-QL queries with the same join and selection signatures

and related company profiles for stocks with price greater than 90 and 100, respectively. The query plan for the query on the left of Figure 2.1 is shown in Figure 2.2. The plan for the query on the right of Figure 2.1 is the same as that in Figure 2.2 except that the constant in the selection predicate is 100 instead of 90.

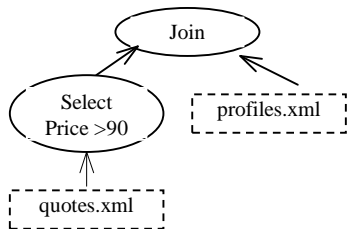


Figure 2.2: Query plan for the first query in Figure 2.1

In general, a range-selection predicate on one attribute can be either “<” or “>”. In our discussion, we consider grouping queries that have the same operator, either “<” or “>”, but not both. One simple way to obtain

this effect is to divide the selection predicates on an attribute into two groups, where one group has only “<” predicates and the other only “>” predicates.

### 2.1. Groups and Expression Signatures

Groups are created for existing queries according to their *expression signatures*, which represent similar structures among the queries. Groups allow the common parts of two or more queries to be shared, with each individual query in a query group sharing the results from the execution of the group plan. In our approach, both selection and join signatures are considered for grouping.

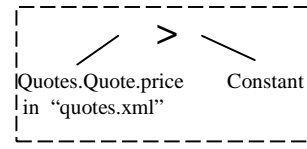


Figure 2.3: A range-selection signature for the example queries in Figure 2.1

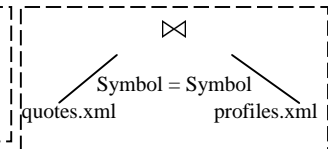


Figure 2.4: A join signature for the example queries in Figure 2.1

A selection expression signature (e.g. Figure 2.3) represents the same syntax structure of a selection predicate with many potentially different constant values (e.g. 90 and 100 in this example). In general, a group consists of three parts: a *group signature*, a *group constant table* and a *group plan*. The *group signature* is the common expression signature of all queries in the group. The selection expression signature for the example above is given in Figure 2.3. The *group constant table* contains the distinct signature constants of all the queries

in the group (along with the names of corresponding intermediate files) and is stored as an XML file. For the example above, (90, file\_i) and (100, file\_j) are stored in this table (Figure 2.5). The *group plan* is the query plan shared by all queries in the group. Since the result of the shared computation contains results for all the queries in the group, the results must be filtered and sent to the correct intermediate files for further processing. NiagaraCQ performs filtering by combining a special *Split* operator with a Join operator on the input of original selection operator (e.g. “quotes.xml” in this example) and the constant table to replace the evaluation of selections of queries in this group. For the example above, the join predicate in the group plan would be “price > Constant\_values” (Figure 2.5), where Constant\_values is an attribute in the constant table used for storing the

distinct constant values. The *Split* operator distributes each result tuple of the Join operator to its correct intermediate file based on the intermediate file name in the tuple (obtained from the Constant Table). Queries with the same constant value also share the same intermediate file. This feature can significantly reduce the number of intermediate files.

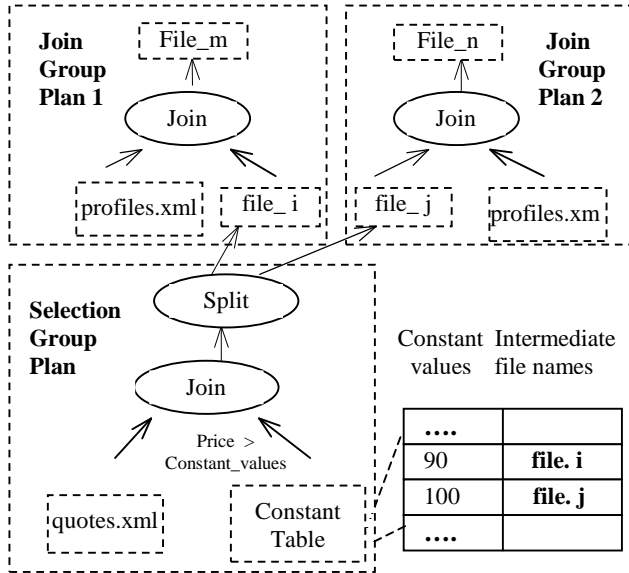


Figure 2.5: Groups created for queries in Figure 2.1 (PushDown)

A join signature represents a join operator on two inputs. A join group is much simpler than a selection group since a join group does not contain any constant values. All queries in a join group have the same join predicate and share the same computation. For the example above, two join groups are created on each intermediate file (*file\_i* and *file\_j*) from the selection group and “profiles.xml” (Figure 2.5).

## 2.2. Query-Split

Our incremental group optimization scheme employs a **query-split** scheme. For illustration purpose, Figure 2.6 shows the query plan of another example query. This query retrieves stock quotes and related company profiles for stocks with prices greater than 90 and groups them by the industry field. Assuming the groups (shown in Figure 2.5) have been created for the queries in Figure 2.1 in the system, when the query in Figure 2.6 is submitted, the group optimizer traverses its query plan bottom up attempting to match its expression signatures with the signatures of existing groups. The selection signature of the new query matches the signature of the selection group in Figure 2.5. The group optimizer then breaks the query plan (Figure 2.6) into two parts. The lower part of the query is removed. The upper part of the query is added onto the group plan. Since 90 is already in the constant table, the associated intermediate file (*file\_i*) is used as the input of the upper part of the query. In the case that the

signature of the query does not match any group signature, a new group will be generated for this signature. This optimization process continues with the remainder of the query tree until the entire query has been analyzed. For the example above, the remainder query plan

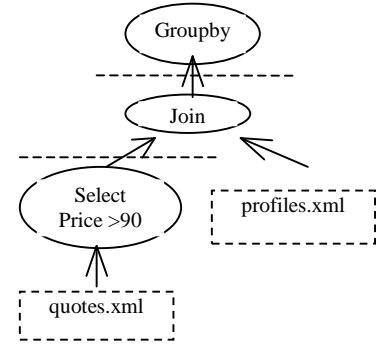


Figure 2.6: Query plan of another example query

will match the join group 1 shown in Figure 2.5 and is split into two parts again (Figure 2.6). The *groupby* operator then becomes an independent query that takes the *file\_m* as its input.

## 2.3. Incremental Evaluation of Grouped Queries

An incremental evaluation approach is used to evaluate grouped continuous queries. Continuous queries are triggered for execution by update events on their data sources. An update event may include multiple data changes. Since frequently only a small portion of each file gets updated, the incremental evaluation of continuous queries can save a significant amount of computation. Another advantage of incremental evaluation is that it avoids repetitive evaluation and only new results are returned to users. For each file (base data file or intermediate file) on which continuous queries are defined, NiagaraCQ maintains a “delta file” that contains recent changes. NiagaraCQ calculates the changes to a source XML file using two snapshots of the file, a stored copy and a newly updated copy. For intermediate files, the outputs from the *split* operators are directly written to the delta files. Whenever possible, queries are run over the delta files. Generally, delta files can be discarded immediately after the queries that use them as inputs have been executed. In some cases the complete files must be used, e.g., incremental evaluation of join operators. Intermediate files, whose complete contents are required by some join operators, must be materialized and kept consistent with respect to data changes on the base data sources. The total query processing cost includes the costs of both performing the necessary computations plus the cost of maintaining the intermediate files as updates to the base files are performed.

While Figure 2.5 illustrates the flow of data at query optimization time, the scenario at query execution time is slightly different. When “quotes.xml” is updated, “Δquotes.xml” is actually used for evaluating the selection group plan. The output from the split operator is written to the delta intermediate files Δ*file\_i* and Δ*file\_j*. The

complete contents of file\_i and file\_j must be retained as they must be joined with “Δprofiles.xml” whenever “profiles.xml” is modified. Thus, Δfile\_i and Δfile\_j must be merged with file\_i and file\_j respectively.

### 3. Incremental Group Optimization Strategies

In this section, the examples from Figure 2.1 are used to illustrate the main ideas of *PushDown* and *PullUp* when incrementally grouping queries with both selection and join operators.

#### 3.1. PushDown

In this method, selection operators are pushed below the join operator. Queries are first grouped by their selection signatures with the output tuples of the selection group stored into multiple intermediate files using our query-split scheme. Multiple join groups are then created on these intermediate files along with the second input to the join operators. This method can be formally represented as

$$(\sigma_1 R \bowtie S) \cup (\sigma_2 R \bowtie S) \cup \dots \cup (\sigma_n R \bowtie S)$$

The group plans generated by applying the PushDown method for the two queries in Figure 2.1 are shown in Figure 2.5. A selection group is first created for the expression signature in Figure 2.3 and two intermediate files — file\_i and file\_j — are allocated. File\_i and file\_j will store tuples that have stock prices above 90 and 100, respectively. Two join groups are then created: one on file\_i and “profiles.xml”; and the other on file\_j and “profiles.xml”.

When the selection predicates overlap, the intermediate query results of the selection group will contain duplicates. Consequently, redundant join operations may be performed between these overlapping intermediate files and the other input file, wasting additional I/O and CPU resources. For example, if a large number of stocks have prices greater than \$100, then there will be a significant amount of overlap between the predicates price > \$90 and price > \$100. When “quotes.xml” is modified, stocks with prices above \$100 will be duplicated in both “Δfile\_i” and “Δfile\_j” and joined with “profiles.xml”. In addition, the duplicate data changes in “Δfile\_i” and “Δfile\_j” must be merged with file\_i and file\_j respectively to keep file\_i and file\_j up-to-date. Similarly, when “profiles.xml” is modified, “Δprofiles.xml” must be joined with overlapping intermediate files file\_i and file\_j. When grouping queries with selection predicates that do not overlap (e.g. stock\_name=”INTC” and stock\_name=”MSFT”), this problem will not occur. However, PushDown still suffers

from the high overhead associated with computing many joins.

The main advantage of this method is that when the number of distinct constant values in the selection signature is small, PushDown avoids computing the entire join between quotes.xml and profiles.xml. Another very important, but less obvious advantage, is that since the data changes are usually confined to only a small portion of the entire data file, the filtering step of the grouped selection may avoid triggering most of the upper level join groups for execution.

#### 3.2. PullUp

An alternate solution is to pull up selections. In this method, the common join operation will be computed before evaluating the different selection predicates. Different selection groups can then be created on the single intermediate file of the join group. This approach can be represented by

$$\sigma_1 (R \bowtie S) \cup \sigma_2 (R \bowtie S) \cup \dots \cup \sigma_n (R \bowtie S)$$

The group plan for the two queries in Figure 2.1 is shown in Figure 3.1. A join group on “quotes.xml” and “profiles.xml” is first created for the expression signature shown in Figure 2.4 and the join result is stored in one intermediate file (file\_k). A selection group is then created on this intermediate file (file\_k). File\_m and File\_n will store tuples with stock prices above 90 and 100, respectively. At query execution time, when “quotes.xml” is updated, “Δquotes.xml” is joined with “profiles.xml”. The results of the join are written to the single delta intermediate files Δfile\_k. The complete contents of file\_k must be retained as they are required when a new entry is

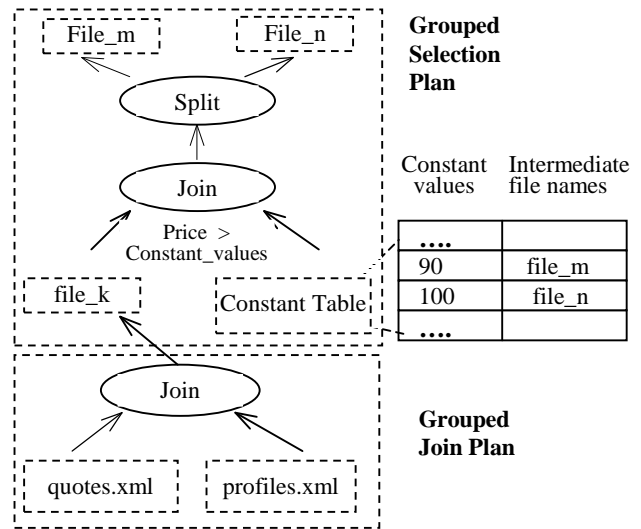


Figure 3.1: Join signature is grouped before selection signature (PullUp)

inserted into the constant table in the selection group. This may happen when a new query with the same expression signatures as those of our example queries (Figure 2.1) but with a new constant in the selection predicate is submitted. Thus,  $\Delta$ file\_k must be merged with file\_k.

In the PullUp approach, only one join group and one selection group is created, since the single intermediate file of the materialized join results allows the remaining of the example queries to have the same selection signature after the queries are merged with the join group. This approach avoids the problem of repetitive joins. Furthermore, PullUp only needs to maintain a single intermediate file, while PushDown generally must maintain multiple overlapping intermediate files.

The main disadvantage of this approach is that since no selection predicates are applied before the join, some tuples that would have been eliminated by the selection predicates end up being joined. If selection operators are blindly pulled over all the join operators in a query with multiple join operators, a very large intermediate file is likely to be produced. In addition, the additional overhead for maintaining the intermediate file (file\_k) will be incurred whenever “profiles.xml” is modified. With the PushDown strategy, changes to “profiles.xml” do not affect the intermediate files.

Note that both PushDown and PullUp produce identical result files.

### 3.3. Filtered PullUp

To overcome the limitations of the PullUp approach, a selection predicate with the union of the selection predicates of the selection group can be inserted before the join operator in the join group plan.

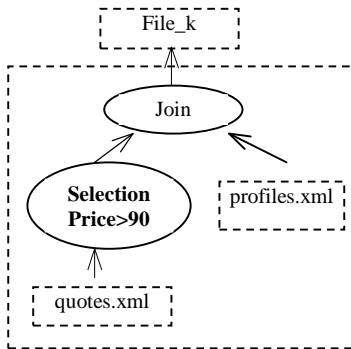


Figure 3.2: Revised Join group plan in filtered PullUp. (Note that the top selection group plan is the same as in Figure 3.1 and is not shown here)

would be inserted before the join operator in the join group plan (Figure 3.2). This optimization can reduce the size of the materialized join result, thus reducing both the I/O cost and the CPU cost of the PullUp method. We call this strategy “Filtered PullUp”. One complexity of this strategy is that the union of the selection predicates may need to be dynamically modified when a new query is added (e.g. in the above example, a query similar to the queries in Figure 2.1 but with price>70) or an existing query is removed

## 4. Analytical Models

In this selection, we present analytical cost models of the PushDown and PullUp strategies, using the example queries presented in Figure 2.1. We have also developed models for queries with equality selection predicates shown in Figure 4.1 but due to space limitations, we only show the formulas for the range selection predicates, while we will present results for both types of queries. For simplicity, we use R and S to represent the names of XML files, quotes.xml and profiles.xml respectively. Unless specified explicitly, we assume that the join attribute has no duplicate values in R and S.

```

Where <Quotes><Quote><Symbol>$s</><Industry>$i</></>
element_as $g </> in “quotes.xml”, $i = “Retail”
<Companies><Company><Symbol>$s</></>
element_as $t </> in “profiles.xml” construct $g, $t
  
```

Figure 4.1: An XML-QL query that retrieves stocks in Retail industry and related company information

The total cost of each strategy includes the cost of query evaluation and the cost of maintaining the persistent intermediate files. For simplicity, we keep the size of R and S constant by assuming half the changes are insert operations and the other half are deletes. An update operation can be treated as a delete operation followed by an insert operation. Assuming the ratio of updates to S versus R is F, we estimate a weighted value of the costs of query processing when either R or S is modified.

Our cost formulas model both I/O and CPU costs. Assuming there is enough memory to be able to hold both inputs to the join, processing a join requires only one scan of its input files. We also assume that data files can be cached in memory and shared by multiple queries. This assumption significantly reduces the I/O cost for both the PushDown and the ungrouped approaches, since a large number of queries may need to scan the same data inputs simultaneously in these two methods. We use the number of predicate evaluations as the metric for CPU cost since this number is more or less independent of the actual implementation and reflects the CPU cost relatively well. Two physical join methods, symmetric hash join (SH) [WA91] and symmetric nested loop (NL) join, are assumed in our formulas. SH and NL are used for equijoins and non-equijoins respectively in our models. The reason that these two algorithms are used is that these two algorithms have been implemented in Niagara and we want to be able to validate the cost formulas developed in this section with experimental results to be presented in next section.

The parameters used in the cost models presented below are tabulated in Table 4.1 along with their default values measured from our system.

Symbol Names	Description	Dflt. Values
R,S	XML files	
$\Delta R, \Delta S$	delta files for R, S	
$ R ,  S $	number of pages in R, S, assuming $ R  =  S $	300
$\ R\ , \ S\ $	number of tuples in R, S (= $ R  * \text{PageSize} / \text{TupleSize}$ )	5000
PageSize	size of a page in bytes	4000
TupleSize	size of a tuple in bytes	240
CtblEntry Size	Size of a constant table entry in bytes	20
IO	Cost in milliseconds(ms) of a disk I/O	2ms
Hc	CPU cost of calculating a hash value and constructing a hash table entry for a tuple	0.01ms
Cc	CPU cost of evaluating a predicate (ms)	0.36ms
N	total number of queries	1000
M	distinct constant values in a selection predicate	100
F	ratio of update event frequency on S/ update event frequency on R	1
K	percentage of upper level join groups triggered for execution in PushDown	100%
$\gamma_i$	the selectivity of a selection predicate	
$\gamma_u$	selectivity of the union of all selection predicates in a selection signature	0.8
$\gamma$	sum of selectivities of M distinct predicates in a selection signature, domain [0,M]	
$\delta_r$	join selectivity of $\Delta R$ join S (number of tuples in $\Delta R$ join S / $\  \Delta R \  * \  S \ $ )	0.0002
$\delta_s$	join selectivity of R join $\Delta S$ (number of tuples in R join $\Delta S$ / $\  R \  * \  \Delta S \ $ )	0.0002

Table 4.1: Parameters of cost models

#### 4.1. PushDown

For each update event on R, let the data changes contain  $|\Delta R|$  pages. Assume there are M intermediate files corresponding to M distinct selection predicates in the selection group, each with selectivity  $\gamma_i$ , where i ranges from 1 to M. The *constant table* in the selection group thus has M entries. The results from the join of  $\Delta R$  and the *constant table* are written into the corresponding M delta intermediate files, each of which contain  $\gamma_i * |\Delta R|$  pages. The total size of delta intermediate files is  $\gamma * |\Delta R|$  pages,

$$\text{where } \gamma = \sum_{i=1}^M \gamma_i.$$

Next, the M delta intermediate files are separately joined with S and the join results are written to disk. For simplicity, we assume the join selectivities of these joins

are the same and equal the join selectivity of  $\Delta R$  join S ( $\delta_r$ ). Let the number of tuples in  $\Delta R$  and S be  $\| \Delta R \|$  and  $\| S \|$  respectively, the final results thus contain  $\delta_r * \gamma * \| \Delta R \| * \| S \|$  tuples. If each page can hold  $\text{PageSize} / \text{TupleSize}$  tuples, the final result files would occupy  $\delta_r * \gamma * \| \Delta R \| * \| S \| * \text{PageSize} / \text{TupleSize}$  pages. Note that for simplicity, in this paper, we assume a constant tuple size. Since from our results expressed in the following sections, I/O cost is only a small portion (around 10%) of the entire cost, such an assumption has little impact on our conclusions.

For clarity, let  $|\text{ConstTbl}|$  represent the number of pages occupied by the *constant table*.  $|\text{ConstTbl}|$  equals  $M * \text{CtblEntrySize} / \text{PageSize}$ . Assuming that the joins only require one scan of both input files, the I/O cost of evaluating these queries is

$$\begin{aligned} & (|\Delta R| + |\text{ConstTbl}|) * IO \quad // \text{read } \Delta R \text{ and the } \text{constant} \\ & \quad \text{table in the selection group} \\ & + \gamma * |\Delta R| * IO \quad // \text{write results from the join of } \Delta R \text{ and} \\ & \quad \text{the } \text{constant table} \text{ into M delta intermediate files} \\ & + (\gamma * |\Delta R| + |S|) * IO \quad // \text{read M delta intermediate} \\ & \quad \text{files and S, assuming one in-memory copy of S can} \\ & \quad \text{be shared by the M joins!} \\ & + \delta_r * \gamma * |\Delta R| * |S| * (\text{PageSize} / \text{TupleSize}) * IO \quad // \text{write final} \\ & \quad \text{results of the joins of delta intermediate files and S} \end{aligned}$$

Assuming the number of tuples in R is  $\| R \|$ , the intermediate files each contain  $\gamma_i * \| R \|$  tuples. A corresponding delta intermediate file contains  $0.5 * \gamma_i * \| \Delta R \|$  inserts and  $0.5 * \gamma_i * \| \Delta R \|$  deletes. Changes in the delta intermediate files must be merged with their corresponding intermediate files. The inserts in a delta intermediate file can be directly appended to the end of its associated intermediate file. In order to delete tuples, the pages that contain the deletes, however, must be fetched into memory and written back to disk<sup>1</sup>.

The I/O cost for intermediate file maintenance is given by

$$\begin{aligned} & 0.5 * \gamma * |\Delta R| * IO \quad // \text{append "inserts" to the end of} \\ & \quad \text{intermediate files} \\ & + 2 * \sum_{i=1}^M \text{Yao}(\gamma_i * \| R \|, \gamma_i * \| R \|, 0.5 * \gamma_i * \| \Delta R \|) * IO \end{aligned}$$

<sup>1</sup> The formula Yao(n, m, k) [Yao77] is used to estimate the number of pages visited when accessing k tuples from n tuples of a file that occupies m pages. When the number of tuples in a page is large (e.g.  $\text{PageSize} / \text{TupleSize} > 10$ ), a good approximation is  $m * (1 - (1 - 1/m)^k)$  [Car75]. Note that in our formula, parameters in Yao(n,m,k) are set as follows: n is  $\gamma_i * \| R \|$ , m is  $\gamma_i * \| R \|$  and k is  $0.5 * \gamma_i * \| \Delta R \|$ .

//read and write pages, which contain “deletes” of intermediate files

The total I/O cost is the sum of the cost of query evaluation plus the cost of keeping the intermediate files consistent.

The CPU cost includes the cost of processing the non-equi-join between  $\Delta R$  and the *constant table* in the selection group and the cost of processing the equi-joins of intermediate files and S in each of the M join groups. Assume that the nested loop join algorithm (NL) is used for the non-equi-join in the selection group, that  $\Delta R$  contains  $\|\Delta R\|$  tuples, and that the constant table contains M entries.  $\|\Delta R\| * M$  comparisons are performed for processing the non-equi-join. Furthermore, assume symmetric hash join (SH) is used for the equi-join in the join groups. For each tuple from either input, the symmetric hash join algorithm first computes the hash value of the join attribute in the tuple and then it inserts the tuple into the hash table on its side; the join algorithm then probes the other-side hash table for possible matches. For each input tuple, let the cost for the first step processing be  $H_c$ . We assume the cost of the probing phase is the cost for one comparison ( $C_c$ ) times the number of matches performed. Since we assume that join attribute values are unique in both R and S, only one comparison is performed in our models. The cost for processing a tuple in a symmetric hash join (SH) thus is  $H_c + C_c$ . From previous discussions, we know that each delta intermediate file contains  $\gamma_i * \|\Delta R\|$  tuples. Thus, the cost of joining S with M delta intermediate files is

$$\sum_{i=1}^M (\gamma_i * \|\Delta R\| + \|S\|) * (H_c + C_c), \text{ which can be easily transformed to a simpler format } (\gamma * \|\Delta R\| + M * \|S\|) * (H_c + C_c), \text{ where } \gamma = \frac{\sum \gamma_i}{M}.$$

The CPU cost thus consists of

$$\begin{aligned} & \|\Delta R\| * M * C_c \quad //\text{process the NL join between } \Delta R \text{ and the } \textit{constant table} \\ & + (\gamma * \|\Delta R\| + M * \|S\|) * (H_c + C_c) \quad //\text{process the SH joins between S and M delta intermediate files} \end{aligned}$$

For each update event on S,  $\Delta S$  is joined with the M materialized intermediate files and the join results are written to disk. There are no changes to the materialized intermediate files in this case. The I/O cost and the CPU cost can both be derived similarly as the case when R is modified. The I/O cost is

$$\|\Delta S\| + \gamma * \|R\| + \delta_s * \gamma * \|R\| * \|\Delta S\| * (\text{PageSize} / \text{TupleSize}).$$

$$\text{The CPU cost is: } (\gamma * \|R\| + M * \|\Delta S\|) * (H_c + C_c).$$

Finally, we calculate a weighted value of the I/O cost for one update event on R and the I/O cost for F update events on S. A weighted-CPU cost can be obtained similarly.

In the analysis above, we assume that all M intermediate files are modified and thus all M upper-level join groups are triggered for execution. In reality, since only a small amount of data is modified at a time, generally only a subset of the M intermediate files will be affected by each update event. Consequently only a subset of the upper level queries will be executed. This filtering feature is very important to the PushDown method since it could reduce the overall cost of PushDown significantly. We use a parameter K to represent the percentage of upper-level join groups that are triggered for execution. Thus, the new cost formula with respect to K is to adjust the cost of the upper level join processing with a factor K.

## 4.2. PullUp

For each update event on R,  $\Delta R$  is first joined with S and the results are written to the single delta intermediate file, whose size is  $\delta_r * \|\Delta R\| * \|S\| * (\text{PageSize} / \text{TupleSize})$  pages. The delta intermediate file is then joined with the *constant table* in the selection group and the final results are written to disk. Since the final results are the same whatever PushDown or PullUp is used, the total size of the final results thus is  $\delta_r * \gamma * \|\Delta R\| * \|S\| * (\text{PageSize} / \text{TupleSize})$  pages (obtained from Section 4.1).

If  $\|\text{ConstTbl}\|$ , the size of the constant table in pages, equals  $M * (\text{CtblEntrySize} / \text{PageSize})$ , then the I/O cost for query evaluation is given by:

$$\begin{aligned} & (\|\Delta R\| + \|S\|) * IO \quad //\text{read } \Delta R \text{ and S in the join group} \\ & + \delta_r * \|\Delta R\| * \|S\| * (\text{PageSize} / \text{TupleSize}) * IO \quad //\text{write join results of } \Delta R \text{ and S into a single delta intermediate file} \\ & + (\delta_r * \|\Delta R\| * \|S\| * (\text{PageSize} / \text{TupleSize}) + \|\text{ConstTbl}\|) * IO \quad //\text{read the delta intermediate file and the } \textit{constant table} \text{ in the selection group} \\ & + \delta_r * \gamma * \|\Delta R\| * \|S\| * (\text{PageSize} / \text{TupleSize}) * IO \quad //\text{write the results of the join of the delta intermediate file and the } \textit{constant table} \end{aligned}$$

Assume the join selectivity of R join S is  $\delta_r$ , and let the number of tuples in R and S be  $\|R\|$  and  $\|S\|$ , respectively. The single intermediate file contains  $\delta_r * \|R\| * \|S\|$  tuples. Its corresponding delta intermediate file contains  $\delta_r * \|\Delta R\| * \|S\|$  tuples, including  $0.5 * \delta_r * \|\Delta R\| * \|S\|$  inserts and  $0.5 * \delta_r * \|\Delta R\| * \|S\|$  deletes. Changes in the delta intermediate file must be merged with the intermediate file and the cost can be derived similarly as that in the previous section.

The I/O cost of intermediate file maintenance is given by

$0.5 * \delta r * |\Delta R| * |S| * (PageSize / TupleSize) * IO$   
 //append “inserts” to the end of the intermediate file  
 $+ 2 * Yao(\delta r * \|R\| * \|S\|, \delta r * |R| * |S| * (PageSize / TupleSize), 0.5 * \delta r * \|\Delta R\| * \|S\|) * IO$   
 //read and write pages of the intermediate file that contain “deletes”

The CPU cost consists of:

$(\|\Delta R\| + \|S\|) * (Hc + Cc)$  //process the SH join of  $\Delta R$  and  $S$  in the join group  
 $+ \delta r * \|\Delta R\| * \|S\| * M * Cc$  //process the NL joins of the delta intermediate file and the *constant table* in the selection group

For an update event on  $S$ , the I/O cost and the CPU cost can be derived very similarly to the case that  $R$  is modified and we omit the derivations and the formulas in the paper. Since the single upper level join is always triggered for execution no matter what data changes occur in PullUp, we do not consider  $K$  in PullUp.

### 4.3. Filtered PullUp

In this method, a selection operator with a selectivity of the union of all selection predicates in the selection group ( $\gamma_u$ ) is placed before the join of  $R$  and  $S$  in the join group in PullUp (Figure 3.2). The I/O and CPU cost formulas in this method are similar to those derived in PullUp, except that the number of tuples from  $R$  or  $\Delta R$  involved in the join is reduced by a factor  $\gamma_u$ . Thus, the size of the intermediate result file is also reduced. One observation is that PullUp can be treated as a special case of *filtered PullUp* with  $\gamma_u$  set to 1.

Note that additional CPU operations are required to evaluate this selection predicate compared with PullUp. For range-selection predicates, because each group consists of only ‘>’ or ‘<’ operator, it is easy to see that the union of the selection predicates is the weakest predicate in the group, which can be evaluated using a single comparison for each input tuple. Since this additional cost is negligible compared to the CPU time spent doing the join, it is not included in our cost formula. For equality predicates, checking the union of selection predicates may consume more time than evaluating a single predicate. However, efficient methods can be designed to bridge the gap (e.g. using a bitmap or a hash index to represent all the constants in the equal predicates).

### 4.4. Ungrouped Case

Since Niagara allows documents to be shared by queries running simultaneously, the I/O cost in the

ungrouped case is a single scan of both data inputs for update events on either  $R$  or  $S$  plus the cost of writing out the final results. The cost of writing out the final results is  $N/M$  times the cost of the grouped cases, since each query writes its output file independently. We omit the formulas in this paper.

For each update event on  $R$ , tuples in  $\Delta R$  are first evaluated against the selection predicate. The CPU cost for processing the selection is  $\|\Delta R\| * Cc$ . Tuples satisfying the selection predicate are then joined with  $S$ . Assuming the selection selectivity of a query is  $\gamma_i$ , the number of tuples involved in the join with  $S$  is  $\gamma_i * \|\Delta R\|$ . The cost of processing the join using symmetric hash join (SH) is  $(\gamma_i * \|\Delta R\| + \|S\|) * (Hc + Cc)$ . The sum of these two costs is the total CPU cost for processing the query. Since  $N$  queries run independently, the total CPU cost is the sum of CPU costs of the  $N$  queries and is given by

$N * \|\Delta R\| * Cc$  //process selections of  $N$  queries  
 $+ N(\gamma / M * \|\Delta R\| + \|S\|)(Hc + Cc)$  //process joins of  $N$  queries using SH algorithms

Note that  $N\gamma / M = \sum_{i=1}^N \gamma_i$ , assuming the constants in the

selection predicates in the  $N$  queries are uniformly distributed among  $M$  distinct values. For update events on  $S$ , the CPU cost can be derived similarly.

### 4.5. Strategy Comparisons

This section examines the performance of the two grouping methods, PushDown and PullUp, by varying several important parameters. The results from the models are shown in Figure 4.2 through Figure 4.9. Our results demonstrate that the performance of the ungrouped method is always tens or hundreds of times worse than the worst case of the grouped strategies, so we do not present the ungrouped results here.

#### Results of range selection (e.g. price>90) queries (Figure 4.2-Figure 4.7)

Assume that there are  $M$  distinct selection predicates in the selection group, let the selectivity of the union of all selection predicates be  $\gamma_u$ , and let the sum of the selectivities be  $\gamma$ . In the experiments, unless specified otherwise, the default values for the three parameters are 100, 0.8 and 20. Under such settings, we simulate an environment in which a fairly large number of queries shown in Figure 2.1 with a high degree of overlap (average selectivity of a selection predicate is 20%). Unless specified,  $F$ , the update frequency to  $S$  versus  $R$ , is 1, and  $K$ , the percentage of upper level join groups triggered for execution in PushDown, is 100%.

We investigate the effectiveness of the optimization, which places a selection operator with the union of all



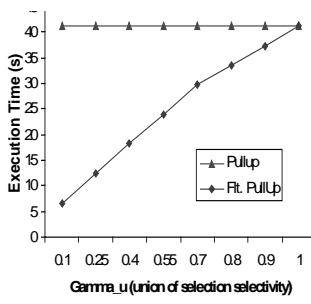


Figure 4.2

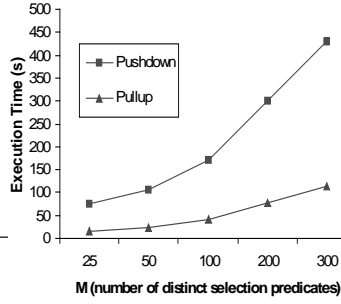


Figure 4.3

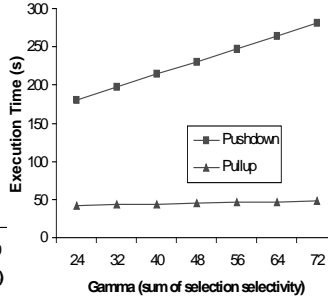


Figure 4.4

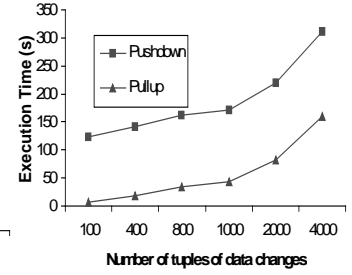


Figure 4.5

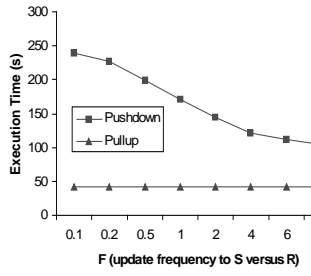


Figure 4.6

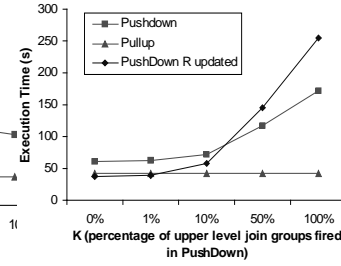


Figure 4.7

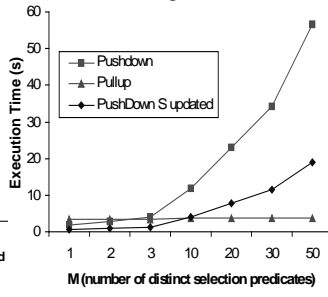


Figure 4.8

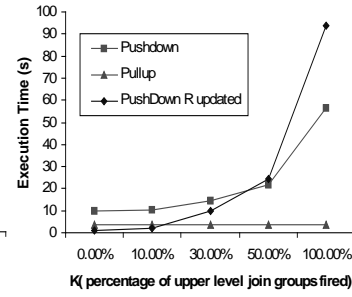


Figure 4.9

selection predicates in the selection group before the join of R and S (Figure 3.2), in the *filtered PullUp* method.  $\gamma$ , the sum of selectivities of M distinct selection predicates is held constant and  $\gamma_u$ , the selectivity of the union of all selection predicates, is varied from 0.1 to 1. Figure 4.2 shows the relative performance of the two strategies. Since the execution time of the PullUp method equals that of *filtered PullUp* with  $\gamma_u = 1$ , *filtered PullUp* is always better than PullUp.  $\gamma_u$  has no influence on the PushDown as long as  $\gamma$  is kept constant. Since we want to compare the performance of PullUp with PushDown regardless of  $\gamma_u$ , in the following study we use PullUp instead.

Figure 4.3 illustrates that the execution time of PushDown increases from 75 seconds to 430 seconds as M, the number of distinct selection predicates, grows from 25 to 300. In this study,  $\gamma$ , the sum of selectivities of M distinct selection predicates is held constant (20). Thus, the size of the intermediate and final query results remains constant. The increase in the execution time is due to the increase of the CPU cost, since the number of joins performed in the PushDown approach is M plus 1. In addition, the CPU cost of performing the join between R and the *constant table* also increases, since the constant table contains M entries. The execution time of the PullUp method is very low (ranging from 15 to 114 seconds). The increase of its execution time is mainly from the CPU cost of joining the single intermediate file with the *constant table*. Due to the fact that  $\gamma/M$ , representing the average selectivity of a selection predicate, can not be less than 1, we can not show the case when M is less than 20.

Assume that M, the distinct predicates in the selection group, is held constant, Figure 4.4 shows that the

execution time of PushDown increases from 180 to 281 seconds as  $\gamma$ , the sum of the predicate selectivities increases. A large  $\gamma$  implies that a large number of tuples are duplicated in the intermediate query results, incurring high I/O and CPU costs for query processing and intermediate file maintenance. The execution time of PullUp method is very low (ranging from 42 to 48 seconds) and only increases slightly as  $\gamma$  increases. This is due to the fact that no duplicated joins are performed with PullUp.

Let the number of data changes for each update event on R or S be represented as  $\|\Delta R\|$  and  $\|\Delta S\|$ , respectively. Assuming  $\|\Delta R\|$  equals  $\|\Delta S\|$ , we vary  $\|\Delta R\|$  from 100 tuples (about 2% of the original data) to 4000 tuples (about 80% of the original data) in this study. Figure 4.5 shows that the execution time of the PullUp strategy remains low (between 7 seconds and 160 seconds). The execution time of PushDown increases from 124 seconds to 310 seconds, since the overlapping intermediate query results incur high I/O and CPU costs.

In Figure 4.6, we vary F, the update frequency to S versus R, to see the effects of different update characteristics on R and S. Recall that in our example queries, one selection is defined over R while no selection is defined over S. One interesting observation is that the execution time of PushDown drops from 240 to 104 seconds, as F increases from 0.1 to 10, because the query processing cost for each update event on S is less than the cost for each update event on R in PushDown. The reason is that in PushDown, the changes on S can be directly joined with the materialized selection results in the intermediate files. Thus, the PushDown method favors the situation that updates on S occur very frequently while R

is modified very infrequently. The cost of PullUp is however, constant since the roles of R and S are symmetric in the formulas of the PullUp method. Another reason is that we assume the size of R and  $\Delta R$  is the same as S and  $\Delta S$ , respectively.

Figure 4.7 shows the execution time of PushDown drops from 172 to 61 seconds as K, the percentage of upper level join groups triggered for execution in PushDown, decreases from 100% to 0%. The execution time of PullUp remains relatively constant since the single upper level join is always triggered for execution no matter what data changes occur. Even when K equals 0, the execution time of PushDown is still longer than PullUp. This is because the execution time plotted in the figure is an average of execution time when either R or S is updated. Since there are 100 groups in this case and data changes on S must be joined with all of them, the average cost is still higher than PullUp. In Figure 4.7, we also show the execution time of PushDown, when only R is modified. We can see the total execution time of PushDown is less than PullUp when less than 2% of the upper level join groups are triggered for execution. Two interesting observations can be drawn from this figure. First, PushDown can be better than PullUp, even in an environment where a large number of highly overlapped continuous queries exist. Second, PushDown favors the situation that R, on which selection groups are created, is modified more frequently than S when only a small percentage of upper level join groups are triggered for execution. This result complements the observation from Figure 4.6 that the PushDown method favors frequent changes on S, on which no selection groups are defined, if all upper level queries are triggered for execution.

In addition, varying the total number of installed continuous queries, N, has no direct effect on the grouped cases, since the performance of these approaches is controlled by the number of groups created, M, and not N. However, as N increases, the cost of non-grouped approaches will increase proportionally.

#### **Results of equality selection (e.g. industry="Retail") queries (Figure 4.8-Figure 4.9)**

We also developed cost models for example queries in Figure 4.1 and conducted a similar performance study (Figure 4.8- Figure 4.9). These queries contain an equality selection predicate on the *Industry* attribute, which has 500 distinct values, yielding a selectivity factor of 1/500. Thus as M, the number of distinct selection predicates increases, the sum of the selectivities,  $\gamma$ , which is  $M/500$ , will increase linearly.

Figure 4.8 shows as M increases from 1 to 50, the execution time of PushDown increases from 1.7 to 56 seconds because the number of joins performed is  $M+1$ . The execution time of PullUp is around 4 seconds, since

the CPU cost of joining the single intermediate file with the *constant table* increases very slightly. The execution time of PushDown when only S is updated is lower than the average time of PushDown when both R and S are updated. This implies that PushDown favors more frequent changes to S, on which no selection groups are created, than R, on which selection groups are created, if all upper level join groups are triggered for execution. On the other hand, Figure 4.9 shows that when K, the percentage of upper level join groups triggered for execution, is small, frequent changes over R are favored in PushDown (M=50 in this experiment).

## **5. Experimental Evaluation**

In the following experiments, we experimentally evaluate PushDown, PullUp and filtered PullUp using the settings for Section 4 and compare the performance results with the results derived in Section 4. The experiments were conducted on a Pentium III 800MHz PC with 256M of RAM, 9GB of Hard Disk, running Linux4.2.

Our experiments were run against a database of stock information consisting of two XML files, "quotes.xml" and "profiles.xml". "Quotes.xml" contains stock information on about 5000 companies. The size of "quotes.xml" is about 1.4 MB. Related company information is stored in "profiles.xml", whose size is about 1.3MB. Data changes on "quotes.xml" and "profiles.xml" are generated artificially to simulate the real stock market and continuous queries are triggered by these changes. We measured the total execution time for each update event on quotes.xml or profiles.xml and calculated a weighted-average of them. We give a brief description of the assumptions that we made to generate "quotes.xml". Each stock has a unique *Symbol* value. The *Industry* attribute takes a value randomly from a set with about 500 values. The *Price* represents the current price of a stock and uniformly distributed among 0 to \$500. Unless specified, for each update event the number of stocks modified in "quotes.xml" is 1000, which is about 280K bytes. Since the time spent calculating changes in two source files is the same for both the grouped and non-grouped approaches, we run our experiments directly against the data changes.

We use two types of queries as shown in Figure 2.1 and Figure 4.1 in our experiments. The equality selection predicate queries are generated using different constants following a uniform distribution over the 500 distinct values on *industry*. However, given an average selectivity of all selection predicates,  $\bar{\gamma}$ , range selection queries follow a uniform distribution over the values with selectivities that range either from 0 to  $2*\bar{\gamma}$  when  $\bar{\gamma}$  is less than 0.5, or from  $2*\bar{\gamma}-1$  to 1 when  $\bar{\gamma}$  is equal to or greater than 0.5.

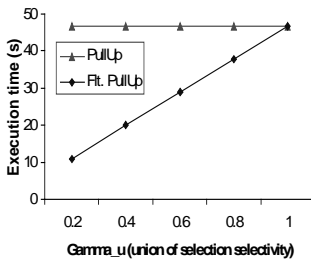


Figure 5.1

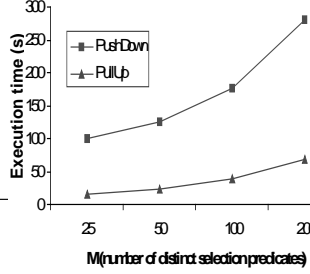


Figure 5.2

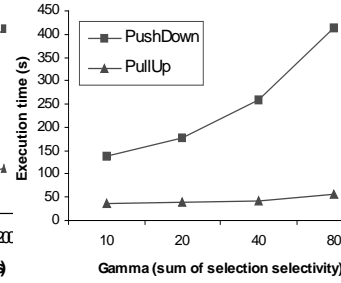


Figure 5.3

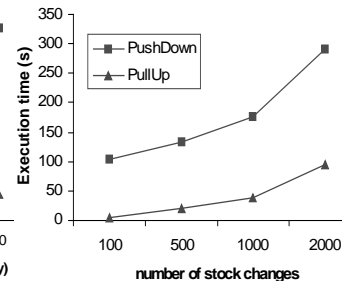


Figure 5.4

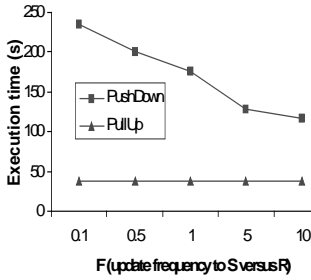


Figure 5.5

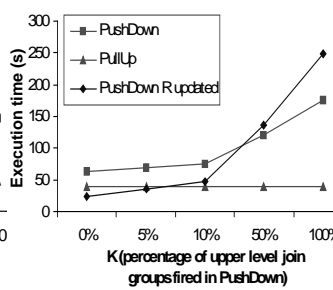


Figure 5.6

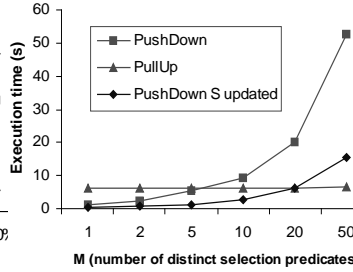


Figure 5.7

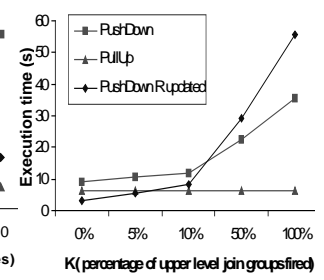


Figure 5.8

We conducted similar experiments to those presented in Section 4. Figures 5.1 through 5.8 show very similar performance results corresponding to the results presented in Figures 4.2 through 4.9.

## 5.1. Summary and Discussion

Overall, the following observations can be drawn consistently from both the analytical models and the experimental studies illustrated by using the simple query examples. PullUp is often much better than PushDown, especially when the number of distinct selection predicates is large and these predicates are highly overlapping. In addition, PullUp is usually much less sensitive to the variance of individual parameters, thus more scalable, than PushDown. PushDown may be favored when the number of distinct selection predicates is small, especially for non-overlapped selection predicates, or under favorable data change workload reflected by parameter K, the number of upper level join groups triggered for execution and F, the ratio of data change frequency on different input.

In general, PullUp could be much more efficient than PushDown for grouping queries containing any number of selections and joins. This is because pushing down selections of queries with multiple joins could result in creating many more intermediate files and thus more join groups than our simple example queries with only one selection and one join operator.

Our results also have some practical applications. In [CDTW00], pushing down selection are used when groups are initially created using incremental group optimization, since at that time, the future workload is not known. If dynamic regrouping is not performed on existing groups, from our findings in this paper, we know that PullUp

could be a better choice for creating initial groups, since the potential performance loss of using PushDown could be much greater than the possible gain obtainable using PullUp. One observation that favors PullUp is that in general, we expect most continuous queries to contain only a few joins (e.g. less than 5). Thus the danger of generating very big join results after pulling up all selections is relatively small. On the other hand, if the best performance is desired for any possible workload, our results show that the group optimizer must dynamically regroup existing queries by considering pulling up selections as the workload changes. Our cost models plus the consideration of pulling up selections can be used by a general purpose multiple query optimizer to perform regrouping.

## 6. Related Work

Our work is closely related to the work of production rule systems (a.k.a. discrimination networks) [For82][Mir87][WH92][NGR88]. RETE [For82] and TREAT [Mir87] are two well-known algorithms used in production rule systems for matching input tokens against installed rules. Our incremental group optimizer generates a RETE-like discrimination network for a group of queries.

Our work is closely related to query optimizations involving materialized views [RSS96][Vis98] [MRSR01]. The intermediate files in our approach are analogous to the additional views in [RSS96]. In addition, we consider incrementally grouping a large number of queries, an idea not considered in [RSS96]. Another related work [Vis98] studied the problem of finding a good plan from among the various alternatives for incremental view maintenance. [MRSR01] applies the multiple-query optimization

techniques in [RSSB00] to materialized views selection and maintenance. Our work is also related to materialized view maintenance [BLT86][GMS93] [Han87] [SR88]. In our method, intermediate files are updated incrementally.

Hellerstein [HS93][Hel94][Hel98] proposed a method called *Predicate Migration* to produce an optimal plan for queries with expensive methods. Our work is related to this work in that we also try to find an optimal plan for queries by considering pulling up selections above joins. However, we focus on optimizing multiple queries with common selection operators in a continuous query environment while his work is on single query optimization with expensive selection predicates.

## 7. Conclusion

In this paper, we design and evaluate alternative selection placement strategies in optimizing a very large number of continuous queries. Two grouping strategies, PushDown and PullUp, in which selections are either pushed below, or pulled above, joins are proposed and investigated. Our study demonstrates that incremental group optimization outperforms the ungrouped approach by factors of up to 350. More interestingly, the results from this paper show that different incremental group optimization strategies can have significantly different performance characteristics. Surprisingly, in our studies, PullUp is often better and achieves an average 10-fold performance improvement over PushDown (occasionally 100 times faster). Furthermore, a revised algorithm of PullUp, termed *filtered PullUp* is proposed, which is able to further reduce the cost of PullUp by 75% when the union of the selection predicates is selective. In addition, detailed cost models, which consider several special parameters, including (1) characteristics of queries to be grouped, and (2) characteristics of data changes, are presented in this paper. Preliminary experiments using an implementation of both strategies show that our models are fairly accurate in predicting the experimental results.

## References

- [BLT86] Jose A. Blakeley, Per-Ake Larson, Frank W. Tompa, "Efficiently Updating Materialized Views," SIGMOD Conference 1986: 61-71.
- [Car75] Alfonso F. Cardenas: Analysis and Performance of Inverted Data Base Structures. CACM 18(5): 253-263 (1975)
- [CDTW00] J. Chen, D. J. DeWitt, F. Tian and Y. Wang, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases," SIGMOD 2000: 379-390.
- [CM86] U. S. Chakravarthy and J. Minker, "Multiple Query Processing in Deductive Databases using Query Graphs," VLDB Conference 1986: 384-391.
- [For82] C. Forgy: RETE: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. Artificial Intelligence 19(1): 17-37 (1982)
- [GMS93] A. Gupta, I. S. Mumick, V. S. Subrahmanian, "Maintaining Views Incrementally," SIGMOD Conference 1993: 157-166
- [Han87] Eric N. Hanson: A Performance Analysis of View Materialization Strategies. SIGMOD Conference 1987: 440-453
- [Hel94] J. M. Hellerstein, "Practical Predicate Placement," SIGMOD Conference 1994: 325-335.
- [Hel98] J. M. Hellerstein, "Optimization Techniques for Queries with Expensive Methods," TODS 23(2): 113-157, 1998.
- [HS93] Joseph M. Hellerstein. Predicate Migration: Optimizing Queries with Expensive Predicates. SIGMOD Conference 1993: 267-276.
- [LPT99] L. Liu, C. Pu, W. Tang, "Continual Queries for Internet Scale Event-Driven Information Delivery," TKDE 11(4): 610-628 (1999).
- [Mir87] D. P. Miranker: TREAT: A Better Match Algorithm for AI Production System Matching. AAAI 1987: 42-47
- [MRSR01] H. Mistry, P. Roy, S. Sudarshan, K. Ramamritham: Materialized View Selection and Maintenance Using Multi-Query Optimization. SIGMOD 2001:307-318.
- [NGR88] P. Nayak, A. Gupta, P. S. Rosenbloom: Comparison of the RETE and TREAT Production Matchers for Soar. AAAI 1988: 693-698
- [RC88] A. Rosenthal and U. S. Chakravarthy, "Anatomy of a Modular Multiple Query Optimizer," VLDB 1988: 230-239.
- [RSSB00] P. Roy, S. Seshadri, S. Sudarshan, S. Bhobe: Efficient and Extensible Algorithms for Multi Query Optimization. SIGMOD Conference 2000: 249-260
- [RSS96] K. A. Ross, D. Srivastava, S. Sudarshan: Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. SIGMOD Conf. 1996: 447-458
- [Sel86] T. Sellis, "Multiple query optimization," ACM Transactions on Database Systems, 10(3), 1986.
- [SR88] J. Srivastava, D. Rotem: Analytical Modeling of Materialized View Maintenance. PODS 1988: 126-134
- [TGNO92] D. Terry, D. Goldberg, D. Nichols, and B. Oki, "Continuous Queries over Append-Only Databases," SIGMOD 1992: 321-330.
- [Vis98] D. Vista: Incremental View Maintenance as an Optimization Problem. EDBT 1998: 374-38.
- [WA91] Annita N. Wilschut, Peter M. G. Apers: Dataflow Query Execution in a Parallel Main-Memory Environment. PDIS 1991: 68-77.
- [WH92] Y. Wang, E. N. Hanson: A Performance Comparison of the RETE and TREAT Algorithms for Testing Database Rule Conditions. ICDE 1992: 88-97.
- [Yao77] S. Bing Yao: Approximating the Number of Accesses in Database Organizations. CACM 20(4): 260-261 (1977).