Determining Essential Statistics for Cost Based Optimization of an ETL Workflow

Ramanujam Halasipuram
IBM Research
India
ramanujam.s@in.ibm.com

Prasad M Deshpande
IBM Research
India
prasdesh@in.ibm.com

Sriram Padmanabhan
IBM Software Group
US
srp@us.ibm.com

ABSTRACT

Many of the ETL products in the market today provide tools for design of ETL workflows, with very little or no support for optimization of such workflows. Optimization of ETL workflows pose several new challenges compared to traditional query optimization in database systems. There have been many attempts both in the industry and the research community to support cost-based optimization techniques for ETL Workflows, but with limited success. Non-availability of source statistics in ETL is one of the major challenges that precludes the use of a cost based optimization strategy. However, the basic philosophy of ETL workflows of design once and execute repeatedly allows interesting possibilities for determining the statistics of the input. In this paper, we propose a framework to determine various sets of statistics to collect for a given workflow, using which the optimizer can estimate the cost of any alternative plan for the workflow. The initial few runs of the workflow are used to collect the statistics and future runs are optimized based on the learned statistics. Since there can be several alternative sets of statistics that are sufficient, we propose an optimization framework to choose a set of statistics that can be measured with the least overhead. We experimentally demonstrate the effectiveness and efficiency of the proposed algorithms.

1. INTRODUCTION

Extract - Transform - Load (ETL) tools are special purpose software artifacts used to populate a data warehouse with up-to-date, clean records from one or more sources. To perform this task, a set of operations should be applied on the source data. The majority of current ETL tools organize such operations as a workflow. At the logical level, an ETL workflow can be considered as a directed acyclic graph (DAG) used to capture the flow of data from the sources to the data warehouse. The input nodes of the graph represent the source record-sets, whereas the output nodes represent the target record-sets that need to be populated. The intermediate nodes represent transformation, cleansing and join activities that reject problematic records and reconcile data to the target warehouse schema. The edges of the graph represent input and output relationships between the nodes.

Unlike SQL, which is declarative in nature, ETL workflows are procedural and specify the sequence of steps to transform the source tables into the target warehouse. Many of the ETL tools in the market today provide support for design of ETL workflows, with very little or no support for optimization of such workflows. The efficiency of the ETL workflow depends to a large extent on the skill and domain knowledge of the workflow designer. This may work well in some situations. However, an ETL workflow is typically designed once and executed periodically to load new data. An ETL workflow that was efficient to start with can easily degrade over time due to the changing nature of the data.

It would be useful to add support for optimization of ETL workflows to ETL engines. Optimization techniques developed for traditional DB systems could potentially be reused. However, there are several challenges specific to ETL that need to be addressed to make this possible. These are summarized below:

- **Variety/Multiplicity of data sources like DBMS, flat files, etc:** A ETL workflow can integrate data from multiple sources. This makes it impossible to push the query into the source systems, since it may need joins of tables across sources. Besides, some of the sources could be flat files or other sources that do not support SQL.
- **Transformation operators:** Many of the operators in an ETL workflow transform data from one form to another, either for cleansing or standardizing into a normal form. These transformation operators are often custom code, the semantics of which may not be known to the optimizer. These operators are essentially black box operations for the optimizer, which makes the optimization extremely challenging.
- **Constraints due to intermediate results:** In an ETL workflow, other than the target table, some intermediate results can also be collected. For example, a common pattern in a join operator is to collect all tuples that do not join with the other table (these are called as *reject links*) into a separate table to aid in diagnostics. Such intermediate results pose additional constraints while reordering a workflow, since the reordering may make it impossible to generate the same set of intermediate results.
- **Non-availability of statistics for cost-based optimization:** Since the source systems are different from the ETL engine, the ETL engine does not have access to the statistics in the source databases. Even if the statistics could be made available, they may be insufficient. The extreme case is when the sources are flat files, since there will be no statistics available

10.5441/002/edbt.2014.29

at all. This makes it impossible to do any kind of cost based optimization.

There have been many attempts both in the industry and the research community to address these challenges and develop optimization techniques for ETL Workflows. However, they have had a limited impact on the ETL tools in the market. We revisit this problem to identify the missing links in the solution. The challenges due to the transformation operators have been largely addressed. However, all the existing techniques assume that statistics are available for cost-based optimization, which is clearly not the case. We address this important gap in this paper, and describe techniques for learning the required set of statistics needed for optimizing the ETL workflow. We exploit the pattern of *design once and execute repeatedly* of ETL workflows to develop an approach for learning the statistics in the initial executions so that future executions can be optimized.

Traditional database systems maintain a set of statistics, often by using histograms, so that it can estimate the cardinalities for various sub-expressions of queries that can be run on the data. Since the set of queries is not known beforehand, the database cannot target the statistics for any specific query. The set of possible statistics for a given table can be quite large. For example, for a table with n columns, there could be 2^n possible multi-attribute distributions, one for each subset of attributes. A multi-attribute distribution on all the attributes is sufficient to compute all the other distributions. However, such a distribution will be very large (likely equal to the database size itself) thus making it very expensive to create and maintain. A common strategy to avoid this is to store only single column distributions and use attribute independence assumption to estimate the multi-attribute distributions needed by the query. This may introduce errors in the estimation, since the attributes may not be independent in practice.

In the *design once execute repeatedly* scenario of ETL, the workflow that is going to run on the data is precisely known. Thus, rather than maintaining the multi-attribute distribution on all attributes, it is possible to figure out a much smaller set of statistics needed to optimize the workflow. These targeted statistics will enable the optimizer to estimate the cardinalities of all possible sub-expressions and to cost all alternative plans. Our proposed system analyzes the workflow to determine a set of statistics that are sufficient to cost any reordering of the flow. For example, consider 3 possible plans for an ETL workflow as shown in Figure 1. In order to be able to cost these alternative plans, one would need to estimate the cardinalities of the sub-expressions: $Orders$, $Product$, $Customer$, $Orders \bowtie Product$, $Orders \bowtie Customer$ and $Product \bowtie Customer$. It can be seen that to estimate these, the set of statistics needed are the distribution of (Product_id, Customer_id) on $Orders$, (Product_id) on $Product$ and (Customer_id) on $Customer$.

Since the ETL engine does not have control on the source databases, these statistics need to be observed by the ETL engine itself. The ETL engine measures these statistics in the initial one or multiple runs and uses them to optimize subsequent runs of the workflow. There could be different sets of statistics that are sufficient to cost any reordering of the flow and the overhead of measuring these statistics could vary widely. In the above example, if the plan 1(a) is executed, the cardinality of $Order \bowtie Product$ can be directly observed. Thus, the only other statistics needed are the distribution of (Customer_id) on $Customer$ and $Orders$ and the cardinality

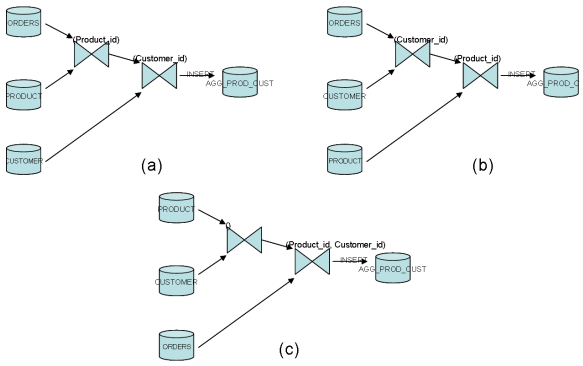

Figure 1: Plans for an ETL workflow

of $Product$. This is likely to be much cheaper in terms of memory overhead since there is no multi-attribute distribution to be measured. We model this as an optimization problem and determine an optimal set of statistics to be measured for any given workflow, such that the overhead of measuring is minimal. In summary, our contributions are as follows:

- A framework for optimization of ETL workflows that are repeatedly executed, when the statistics on the source tables are either unavailable or incomplete.
- An algorithm for selecting an optimal set of statistics to be measured given an ETL workflow, such that any reordering of the workflow can be costed accurately by the optimizer.
- An experimental evaluation to validate the effectiveness of the proposed algorithm.

There is a caveat that the underlying data may change even though the same flow is executed repeatedly. Thus, the statistics gathered in one execution may not be valid future executions. In practice the data changes gradually and thus we assume that the statistics from one execution are valid for the next execution. The whole cycle is repeated in each execution so that the statistics are kept updated with the changing data. Further, it can be noted that though the techniques are explained in the context of ETL workflows, they apply equally to SQL queries which are repeatedly executed and warrant the extra effort of gathering accurate statistics to enable perfect optimization. The rest of the paper is organized as follows. In Section 2, we survey the related work in this area. In Sections 3 and 4, we lay the framework and describe how to determine the statistics required to cost any plan. We further extend this into an optimization framework for determining the optimal set of statistics in Section 5. In Section 6, we describe how to exploit metadata about functional dependencies to reduce the statistics needed. We describe the experimental results in Section 7, discuss possible future work in Section 8 and finally conclude in Section 9.

2. RELATED WORK

There have been many techniques proposed by the research community in the past for cost-based optimization of ETL workflows. Most of the papers [17, 23, 16, 14, 21] focused on conceptual modeling of ETL optimization. [17] models the problem as an state-space search problem and defines operators for generating the search space. [23] delves into the conceptual modeling of the ETL process, while [16] details the mapping of them into logical

ETL processes. [14] touches upon efficient heuristics for logical optimization of the ETL workflows. [21] extended the ETL optimization to physical implementation for the logical counterparts. Recently metrics other than cost have also been considered for optimization. [18] introduces the idea of optimizing ETL workflows for quality metrics such as reliability and scalability. [19] considers optimizing ETL workflows for external interruptions like faults etc. All these ETL optimization strategies assume the availability of statistics necessary in determining the cost of the operator and focus on the process of cost-based optimization using the operator's cost. In contrast, the primary contribution of our work is to address the issue of estimating the operator cost when the input statistics are either missing or incomplete.

Some of the commercial ETL engines support static rule-based optimizations. For example, IBM DataStage has introduced this notion under its Balanced Optimization feature [9], while Informatica [11] also has similar features built into its product. While these techniques do support workflow optimization, the static nature of the rules doesn't take any cost metrics into consideration.

Though the current work is primarily motivated by ETL workflows, there is a lot of work in SQL query optimization area that is somewhat related to this work. Most of the published work regarding SQL query re-optimization can be classified into one of the following two categories:

a) re-optimizations of the current (or ongoing) query; and

b) optimization of future queries

Techniques such as mid-query re-optimization [12], eddies [2], proactive re-optimization [15], query scrambling [22], etc. belong to the first category. Work related to statistics tuning and learning [7, 1, 5, 20, 3, 4, 6, 8] fall into the second category. IBM Learning Optimizer(LEO) [20] explores the idea of using actual cardinalities for adjusting the optimizer estimates. [3, 4] extend the notion of cardinality observation to intermediate sub expressions and introduce a framework for observing the cardinalities of SEs and using them as part of query optimization. The work most relevant to this paper is the pay-as-you-go framework [6], which recognizes that just observing the cardinalities may not help in finding the optimal plan for the current query. The actual cardinalities of many sub-expressions not covered in the current plan are not observed, since the plan is not being altered. They introduce the idea of plan modification to ensure that all the sub-expressions are covered over the different plans. This enables observation of the cardinalities of all the sub-expressions, thus enabling the selection of the optimal plan. Of course, exploring the cardinalities of all the sub-expressions might be an overkill and to strike a balance, XPLUS [8] introduces experts which control the trade-off between exploration of the search space (to determine cardinalities of different sub-expressions) and exploitation of cardinalities of the known sub-expressions. However, in both of these techniques, the only way to determine the cardinality of a sub-expression is to observe it directly. They do not consider observing other statistics that could be used to compute the cardinalities. Thus, to be able to measure all cardinalities of all possible sub-expressions, they would require to run a large number of plans. To address this limitation, our framework generalizes this to observe different kinds of statistics, including cardinalities and attribute distributions. A smaller number of attribute distributions are sufficient to compute the cardinalities of all possible sub-expressions.

Our proposed technique falls into the second category of optimization of future queries, based on observations made in the current run. To the best of our knowledge, this is the first effort to develop a systematic framework that considers alternative sets of statistics for a given query to choose the option with the minimal overhead of observation.

3. STATISTICS COLLECTION FRAMEWORK

3.1 Preliminaries

In this section, we introduce the concepts and terminology used in the rest of the paper. The notations used in the following discussion are listed in Table 1.

Symbol	Description
$\mathcal{E}$	The set of all possible sub-expressions over all the plans for an ETL flow
T_i	The relation T_i corresponding to sub-expression e_i
$T_{i_1,i_2,\ldots i_m}$	$T_{i_1} \bowtie T_{i_2} \bowtie \ldots T_{i_m}$
$\|T\|$	The cardinality of relation T
H^a_T	Histogram on attribute a of relation T
$\|H^a_T\|$	The sum of the values in the histogram, this will be equal to $\|T\|$
$\|a_T\|$	The number of distinct values of attribute a in relation T
$\|a\|$	The domain size of a over all relations
J_{ij}	Join key between T_i and T_j
$\overset{J_{ij}}{\bowtie}$	Join of relations T_i and T_j using J_{ij}
$T_i^{J_{ij}}$	The rows from T_i that satisfy the join predicate J_{ij}
$\overline{T}_i^{J_{ij}}$	The rows from T_i that were rejected by join predicate J_{ij}
$\langle H^a_{T_1} \| H^a_{T_2} \rangle$	For each bucket of the histogram $H^a_{T_1}$, multiply the frequency value with its corresponding frequency value in $H^a_{T_2}$
$\frac{H^a_{T_1}}{H^a_{T_2}}$	For each bucket of the histogram $H^a_{T_1}$, divide the frequency value with its corresponding frequency value in $H^a_{T_2}$
$G(T,a)$	Group by of T on attribute a
$U(T,a)$	A ETL transform operator (UDF) applied to attribute a of T

Table 1: Symbol descriptions

Sub-expressions: Given a ETL workflow, determining an optimal plan based on a cost metric involves identifying different possible re-orderings of the given flow and cost them. Cost-based optimizers use different transformation rules defined by the semantics of the operators to determine alternative orderings of the given flow. These transformation rules define all valid re-orderings of the operators and thus enable the optimizer in generating a search space of candidate plans. Once the candidate plans are identified, operator cost models help the optimizer in determining the cost of the plan. The cost model estimates the cost of each operator based on inputs like the cardinalities of the input relations, CPU and disk-access speeds, memory availability, etc. The most important factors determining the cost of any operator (including the standard select, project, join and group-by operators) are the cardinalities of the inputs. Thus, for a given plan, if the cardinalities of the outputs at all intermediate stages of the plan are determined, the cost of any operator in the plan and therefore the total cost of the plan could

be computed. A sub-plan denotes a subset of the plan till some intermediate stage and a sub-expression (SE) logically denotes the result at an intermediate stage of the plan.

For example, consider the sample plans shown in Figure 1 again. The SEs for the plan 1(a) are $Orders$, $Product$, $Customer$, $Orders \bowtie Product$. A different plan for the same query may produce additional SEs. For example, plan 1(b) will produce the SE $Orders \bowtie Customers$. Thus, in order to be able to cost all possible plans, we need to look at all possible SEs that can be produced in any of the plans. We will denote the set of all possible SEs over all possible plans as $\mathcal{E}$. For simplicity of explanation, the expression corresponding to the complete flow is also included in $\mathcal{E}$. Note that a SE is a logical entity and different plans may produce the same SE in different ways. For example, all the three plans in Figure 1 produce the same SE $Product \bowtie Orders \bowtie Customer$. In general, for a join of n tables $T_1, T_2, \ldots T_n$, considering all possible join orders, the set $\mathcal{E}$ will contain all possible joins, i.e. joins corresponding to each of the 2^n subsets of $\{T_1, T_2, \ldots T_n\}$.

Candidate Statistics Set: To determine some statistics of a SE, other statistics on the composing SEs can be used. A set of statistics that is sufficient for computing a statistic of a SE is defined as a *sufficient statistics set* for that statistic. Further, such a set is minimal if any subset of it is not sufficient. We denote such a minimally sufficient set of statistics as a *candidate statistics set (CSS)* for the statistic. There could be multiple CSSs for a statistic. Note that a possible CSS for a statistic is a set containing that statistic itself. This set is referred to as the the *trivial CSS* for that statistic.

For example, for the statistic $|Orders \bowtie Customer|$, a CSS is $\{H^{Customer\_id}_{Orders}, H^{Customer\_id}_{Customer}\}$, since $Customer\_id$ is the join column and join cardinality can be estimated if the distributions on the join attributes of both the relations are known. The trivial CSS in this case is $\{|Orders \bowtie Customer|\}$. Similarly, for the statistic, $H^{Customer\_id}_{Orders}$, a possible CSS is $\{H^{Customer\_id,Product\_id}_{Orders}\}$ and the trivial CSS is $\{H^{Customer\_id}_{Orders}\}$.

We further define a notion of an *observable statistic*. A statistic is observable in a given plan if it can be observed by instrumenting the plan to collect statistics at the appropriate points. For example, consider the plan shown in Figure 1(a). Both the statistics in the CSS $\{H^{Customer\_id}_{Orders}, H^{Customer\_id}_{Customer}\}$ are observable in this plan, since the plan can be instrumented to build the histograms on $Orders$ and $Customer$, just after the corresponding nodes in the plan. On the other hand, the trivial statistic $|Orders \bowtie Customer|$ is not observable, since $Product$ is joined with $Orders$, before joining with $Customer$.

The estimates will be exact only if the histogram stores a frequency count for each value in the domain. In reality, the estimates will be approximate since histograms bucketize the values and store the average frequency count for each bucket. Currently, we consider only histograms that can accurately estimate the cardinalities. Estimation errors introduced because of approximate statistics are left as part of a future exercise (Section 8).

Problem Statement: Any framework that identifies sufficient statistics to enable cost-based optimization should guarantee that the statistics identified are enough to compute the cost of any SE in the set $\mathcal{E}$ for the given flow. If at least one CSS for each SE in the set $\mathcal{E}$ is available, the cost of any plan for the given flow can be computed, thus enabling the cost-based optimizer to select the best plan. Thus, the goal of the framework can be stated as follows: *given an ETL flow, identify a set of statistics to observe such that it covers at least one CSS for each SE in the set $\mathcal{E}$ corresponding to the flow*.

3.2 The Framework

The overall flow of the ETL optimization process is shown in Figure 2. The process starts with the initial plan, i.e. the workflow defined by the user. The system analyzes the workflow to determine optimizable blocks. For each optimizable block, the set of all possible SEs is determined. The next step is to determine the possible CSS for each SE. A set of statistics is determined such that it contains at least one CSS for each SE. The plan is then instrumented with code to collect these statistics and then run to actually gather the statistics. Based on the collected statistics, the optimizer can now cost alternative plans and the best plan is chosen for future runs of the flow. The entire cycle is repeated periodically, since the underlying data characteristics may be changing. If the data changes sufficiently, a plan that was optimal at one time may no longer be optimal. So it becomes necessary to collect the statistics and re-optimize again. The process can either repeat at each run of the workflow or at some other user defined interval.

Steps 1, 2 and 7 are standard steps for any optimizer. Since they have been covered by prior work on ETL optimization, we just provide a brief description, pointing out some considerations specific to ETL flows. Steps 3 to 6 pertain to identifying and gathering the required statistics and form the focus of this paper. If an ETL engine already has an optimizer module, we can integrate these additional steps into the optimizer flow. We elaborate on the steps in the following sections.

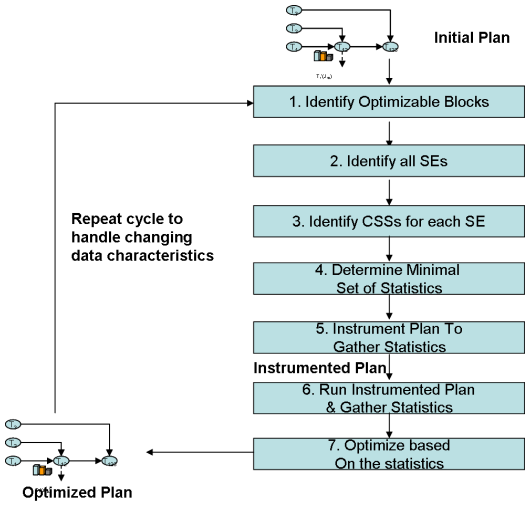

Figure 2: Overview of the Optimization Process

3.2.1 *Identifying Optimizable Blocks*

Due to some constructs in the ETL, it may not be possible to optimize the entire workflow as one unit. The workflow needs to be broken up into smaller units, each of which can be independently optimized. In this step, the system analyzes the workflow to identify the boundaries in the flow across which the the operators cannot be moved for optimization. Specifically, the following conditions need to be checked:

- **Materialized Intermediate Results:** ETL flows often materialize some intermediate results, typically to aid diagnostics or to be used in some other flow. A common example is a reject link that collects the tuples in a relation that do not join with the other relation. Blocking operators such as sort may also need the preceding results to be explicitly materialized. Any point at which an intermediate result is explicitly materialized identifies a block boundary.

- **Transformation Operators:** Another common pattern is the use of operators that transform attribute values from one form to another. Often, the transformation operators do not affect the join re-orderings. However, in some cases, when the operator is applied on an attribute derived from the join of multiple relations $T_1, T_2, \ldots T_n$, and when its result is used in a further join, it forces the relations $T_1, T_2, \ldots T_n$ to be always joined before they join with the rest of the relations. This, in effect, creates a block boundary.

- **Aggregate UDF operators:** UDFs and custom operators are also frequently used in ETL workflows. A custom operator that aggregates its input tuples to produce a smaller number of output tuples is blocking in nature. Since the semantics of the operators is a black box to the optimizer, the safest strategy is to consider it as a block boundary.

Consider the example workflow shown in Figure 3. This workflow will be divided into three optimizable blocks. The first block boundary B_1 is due to the fact that reject link $\overline{T_1}$ is materialized. The second boundary B_2 is due to the UDF transformation that creates a derived attribute c that is a join attribute of the subsequent join with T_4. These boundaries imply that any reordering of joins should respect the block boundaries, for example, T_3 cannot be moved across B_1. The block boundaries reduce the search space for the optimizer since each block can be optimized independently.

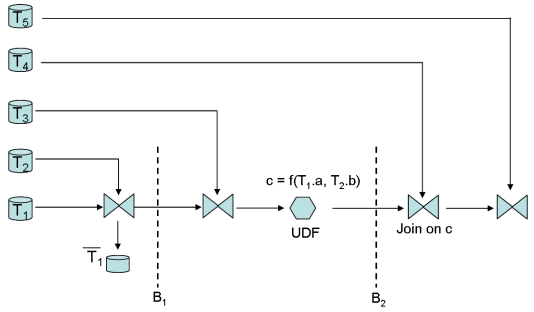

Figure 3: Optimizable Blocks

3.2.2 *Generating Sub-expressions*

The next step is to identify all possible SEs for each optimizable block. The set of possible SEs depends on the semantics of the operators, which determines where the operator can be placed in the flow. For a join on multiple relations, there are many different join orders possible and each join order would generate a set of SEs. For example, for a join on 3 relations, the set $\mathcal{E}$ consists of $\{T_1, T_2, T_2, T_{12}, T_{13}, T_{23}, T_{123}\}$. As a case in point, T_{13} occurs in the join order $(T_1 \bowtie T_3) \bowtie T_2$.

The optimizer may not have support for a few valid transformations. For example, for the initial plan shown in fig 4(a), the reordered plan shown in fig 4(b) is a valid transformation. However, the optimizer may not support such a transformation. The optimizer may also exploit some metadata to avoid generating some plans and reduce the search space. For example, a foreign key join is essentially a *look-up* and the cardinality of the join-result is same as the cardinality of the foreign-key table. In case of a cross product, the cardinality of the join-result is the product of the input cardinalities. Thus, the optimizer may only consider plans that have foreign key joins and no cross products. In general, there is no need to consider SEs from any plan that the optimizer is not going to generate in its search process, since the optimizer does not need to estimate the cost of such plans. To avoid this mis-match, a close integration is required in which only the plans generated by the optimizer are considered for generating the set $\mathcal{E}$.

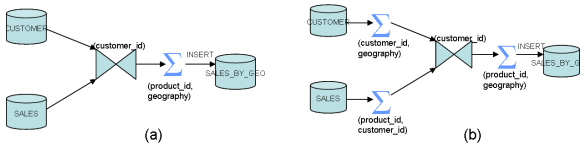

Figure 4: Sample ETL with aggregation

3.2.3 *Generating Candidate Statistics Set*

Once the SEs are determined, the system computes possible CSSs for each of the SE. Each CSS for an SE provides an alternative for estimating the cardinality of that SE. We elaborate on the process of generating CSSs in Section 4.

3.2.4 *Determine Minimal Set of Statistics*

There is a cost associated with observing a CSS in a given flow, which could include the cpu cost and the memory cost for observing the distributions. In this step, a set of statistics is chosen, such that at least one CSS for each SE is covered, and at the same time the cost of observing the statistics is minimal. The details of this process are described in Section 5.

3.2.5 *Instrument Plan to Get Statistics*

The plan has to be instrumented to observe the set of statistics that is chosen by the previous step. Many commercial ETL engines provide a mechanism to plug in user defined handlers at any point in the flow. These handlers are invoked for every tuple that passes through that point. This makes it very easy to plug in code that can observe the required statistics. We consider two main types of statistics:

1. **Cardinality:** The cardinality of any observable SE can be observed by maintaining a simple counter at the corresponding point in the flow. The counter is incremented for each tuple passing through that point. The memory cost of this is the overhead of maintaining one integer (4 bytes) in memory.

2. **Distributions:** The distribution (histograms) of any observable SE can be observed by maintaining a histogram at the corresponding point. For each tuple passing through the point, the attribute corresponding to the histogram is observed and the corresponding histogram bucket is incremented. The memory cost of this is equal to the domain size of the attribute on which the histogram is being built.

3.2.6 *Run Instrumented Plan and Observe Statistics*

In this step, the instrumented plan is executed and the required statistics are gathered. The previous steps ensure that sufficient

statistics are now available for the optimizer to cost any possible plan for the given ETL flow.

3.2.7 Optimize ETL

This step uses traditional cost based optimization techniques to determine the plan with the least cost. Since all the required statistics are already computed, the cost of each alternative plan can be accurately determined. We will not elaborate on this step further, since any existing cost based optimization technique can be used.

4. GENERATING CANDIDATE STATISTICS SET

As mentioned in the previous section, the step of generating CSSs should be closely integrated with the optimizer, since only the plans considered by the optimizer in its search process need to be considered. Optimizers typically use dynamic programming, in which the SEs are incrementally built into larger SEs. For each SE, the optimizer considers alternative plans to compose it from smaller SEs.

DEFINITION 1. **Plan:** *A plan specifies a method of evaluating a SE based on smaller SEs, i.e.* $p_e : op(e_1, \ldots e_k)$*, where* p_e *denotes a plan for SE* e*,* op *is an operator and* $e_1, \ldots, e_k$ *are other SEs. For example, two plans for* $T_{1,2,3}$ *are* $\bowtie\ (T_{1,2}, T_3)$ *and* $\bowtie\ (T_1, T_{2,3})$*. Let* P_e *denote the set of all plans for* e*.*

We assume that we can get the set of all SEs and the plans considered for them from the optimizer, i.e. we can get the set $\mathcal{P} = \{(e, P_e) : e \in \mathcal{E}\}$. The next step is to generate the CSSs for computing the cardinality of each SE, using the rules described next.

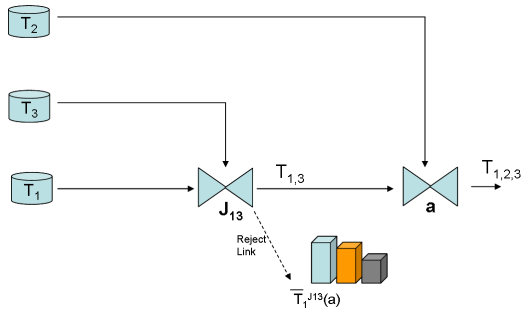

Figure 5: Sample ETL plan

4.1 Rules for generating CSS

Let us consider the problem of determining some statistics of a SE. The types of statistics we consider include cardinality ($|T|$), distinct values for an attribute ($|a_T|$), and distributions (H^a_T). The statistic on a SE may be directly observable if that SE occurs in the plan being executed. In other cases, it could be computed from other statistics, depending on the plan being considered. The semantics of the operator in the plan determines how the output statistics can be computed. In general, to enable estimation over composition of operators, we need to define rules for each type of operator.

DEFINITION 2. **Rule:** *A rule specifies a method of computing a statistic on a SE, based on the statistics of other SEs and the operator being applied. Let* $s_e = (s, e)$ *denote a statistic for SE* e*,* S_e *be the set of all statistics for* e *and* $\mathcal{S}$ *be the set of all statistics over all the SEs. A rule is a function* $P_e \times S_e \rightarrow 2^{\mathcal{S}}$*, i.e. a rule* r *applied to a statistic* s_e *and plan* p_e *determines the set of other statistics* $\{s_{e_1}, \ldots s_{e_k}\}$ *that can be used to compute* s_e*, under plan* p_e*. For example, for the SE* $T_{1,2,3}$ *and the plan* $Join(T_{1,2}, T_3)$*, one possible rule is that the output cardinality can be computed using the input distributions on the join column, i.e.* $|T_{1,2,3}|$ *can be computed using* $\{H^a_{T_{1,2}}, H^a_{T_3}\}$*, where* a *is the join attribute.*

These rules can be applied recursively to generate different CSS for a given a SE. We list the rules for some of the common operators used in ETL below.

4.1.1 Select and Project Operators

The rules for select and project operators are listed in Table 2 and are quite straightforward. The first rule says that the cardinality of a selection can be estimated if the distribution on the selection attribute is known. The second rule specifies that the distribution of an attribute b on the output of a selection on attribute a can be estimated if a joint distribution on (a, b) is known on the input relation. The project operator only selects certain columns, so the output cardinalities and distributions are identical to the input cardinalities and distributions.

Id	Plan	s_e	Inputs
S1	$\sigma_a(T_1)$	$\lvert\sigma_a(T_1)\rvert$	$H^a_{T_1}$
S2	$\sigma_a(T_1)$	$H^b_{\sigma_a(T_1)}$	$H^{(a,b)}_{T_1}$ if $b \neq a$
P1	$\pi_a(T_1)$	$\lvert\pi_a(T_1)\rvert$	$\lvert T_1\rvert$
P2	$\pi_a(T_1)$	$H^b_{\pi_a(T_1)}$	$H^b_{T_1}$

Table 2: Rules for Select and Project

4.1.2 Join Operator

There are multiple ways to estimate the cardinality of a join operator. These are listed in Table 3. The first set of rules (J1 and J2) are derived from the standard technique used by optimizers to estimate join cardinalities. The cardinality of a join can be determined from the distributions on the input tables on the join attribute, by taking a dot product, i.e. $|T_{1,2}| = H^a_{T_1}.H^a_{T_2}$, where a is the join attribute. Similarly, to estimate the distribution on the output of the join, we need a joint distribution on attributes a, b on the table to which b belongs. A matrix multiplication between the two distributions $H^{a,b}_{T_1}$ and $H^a_{T_2}$ will produce the required distribution on the join results $H^b_{T_{1,2}}$, in the case where $b \in T_1$.

Id	Plan	s_e	Inputs
J1	$\bowtie_a (T_1, T_2)$	$\lvert T_{1,2}\rvert$	$H^a_{T_1}, H^a_{T_2}$
J2	$\bowtie_a (T_1, T_2)$	$H^b_{T_{1,2}}$, where $b \neq a$	$H^{a,b}_{T_1}, H^a_{T_2}$ if $b \in T_1$ $H^a_{T_1}, H^{a,b}_{T_2}$ otherwise
J3	$\bowtie_a (T_1, T_2)$	$H^b_{T_{1,2}}$, where $b = a$	$H^b_{T_1}, H^b_{T_2}$
J4	$\bowtie_a (T_1, T_2)$	$\lvert T_{1,2}\rvert$	$H^{J_{13}}_{T_{1,2,3}}, H^{J_{13}}_{T_3}, \lvert\overline{T_1}^{J_{12}} \bowtie T_2\rvert$
J5	$\bowtie_a (T_1, T_2)$	$H^b_{T_{1,2}}$	$H^{J_{13},b}_{T_{1,2,3}}, H^{J_{13}}_{T_3}, H^b_{\overline{T_1}^{J_{12}} \bowtie T_2}$

Table 3: Rules for Join

Rules J4 and J5 are derived from the union-division method, which is a new method proposed by us in order to exploit the observable statistics from the plan to the maximum. For example, if the initial plan was as shown in Figure 5, then SE $T_{1,2}$ is not directly observable. However, $T_{1,2,3}$ is observable, so we try to exploit the

distributions on $T_{1,2,3}$. All the rows that form part of IR $T_{1,2,3}$ would be part of $T_{1,2}$. Rows from T_1 that do not join with T_3 get filtered from $T_{1,2,3}$, whereas they are included in $T_{1,2}$. Thus,

$$\begin{aligned} T_{1,2} &= T_1 \overset{a}{\bowtie} T_2 \\ &= (T_1^{J_{13}} \cup \overline{T}_1^{J_{13}}) \overset{a}{\bowtie} T_2 \\ &= (T_1^{J_{13}} \overset{a}{\bowtie} T_2) \cup (\overline{T}_1^{J_{13}} \overset{a}{\bowtie} T_2) \end{aligned} \quad (1)$$

Thus to compute the cardinality of $T_{1,2}$, we need to compute the cardinalities of $(T_1^{J_{13}} \overset{a}{\bowtie} T_2)$ and $(\overline{T}_1^{J_{13}} \overset{a}{\bowtie} T_2)$. Denoting $(T_1^{J_{13}} \overset{a}{\bowtie} T_2)$ as $T'_{1,2}$, and considering the fact that a join has a multiplicative effect on the distribution of the join attribute,

$$\begin{aligned} T_{1,2,3} &= T'_{1,2} \bowtie T_3 \\ Thus, H^{J_{13}}_{T_{1,2,3}} &= \left\langle H^{J_{13}}_{T^{\square}_{1,2}} | H^{J_{13}}_{T_3} \right\rangle \\ Thus, H^{J_{13}}_{T^{\square}_{1,2}} &= \frac{H^{J_{13}}_{T_{1,2,3}}}{H^{J_{13}}_{T_3}} \end{aligned} \quad (2)$$

Putting them together, we get:

$$\begin{aligned} \left|T'_{1,2}\right| &= \left| H^{J_{13}}_{T^{\square}_{1,2}} \right| \\ &= \left| \frac{H^{J_{13}}_{T_{1,2,3}}}{H^{J_{13}}_{T_3}} \right|, from\ Equation\ 2 \end{aligned} \quad (3)$$

Thus, to compute the cardinality of $T_{1,2}$, we need to observe $H^{J_{13}}_{T_{1,2,3}}$, $H^{J_{13}}_{T_3}$ and $\left|(\overline{T}_1^{J_{13}} \bowtie T_2)\right|$, as mentioned in the rule J4. The rule J5 can be similarly derived. Note that to observe $\overline{T}_1^{J_{13}} \bowtie T_2$, we may need to add an explicit reject link for T_1 after its join with T_3, if it does not already exist, as shown in Figure 5. Though it looks like this technique requires a lot more statistics, it can be cheaper since the number of tuples on the reject link can be small.

4.1.3 Group By Operators

The rules for group-by operator are listed in Table 4. The first rule specifies that the cardinality of the group-by is same as the number of distinct values of the group-by attributes in the input table. The distribution of attributes b on the group-by result can be computed from the histogram on T for attributes a, when $b \subseteq a$. If b is not a subset of a, the distribution does not exist, since b will not be present in the output tuples.

Id	Plan	s_e	Inputs
G1	$G(T,a)$	$\|G(T,a)\|$	$\|a_T\|$
G2	$G(T,a)$	$H^b_{G(T,a)}$	$H^{(a)}_T$ if $b \subseteq a$

Table 4: Rules for Group By

4.1.4 Transformation Operators

Finally, the rules for transformation operators are listed in Table 5. These operators could even be custom user defined functions. Since transformation operators only transform the attributes, they do not affect the cardinality. Thus the cardinality of the output is same as that of the input. The distribution of attributes b on the transformation result is the same as the distribution of b on the input, if $b \neq a$. This is because the transformation leaves b unchanged. If $b = a$, the distribution of output cannot be computed from the input distributions, since it depends on the actual transformation function.

Id	Plan	s_e	Inputs
U1	$U(T,a)$	$\|U(T,a)\|$	$\|T\|$
U2	$U(T,a)$	$H^b_{U(T,a)}$	$H^{(b)}_T$ if $b \neq a$

Table 5: Rules for Transformation operators

4.1.5 Identity Rules

These rules are not specific to any operator, but directly apply to any SE and are listed in Table. Rule I1 specifies that the cardinality of a relation can be computed from a histogram on any set of attributes of that relation, by just adding up the bucket values. Rule I2 specifies that a histogram on attribute set a can be computed from a more fine-grained histogram on attributes (a, b), again by aggregating on the buckets on the b attribute.

Id	s_e	Inputs
I1	$\|T\|$	H^a_T
I2	H^a_T	$H^{a,b}_T$

4.2 Applying the rules to generate CSS

Given an initial workflow, and the plan space generated by the optimizer, we can apply the rules to generate the CSS for the flow. The algorithm is listed in Algorithm 1. W is the workflow for which the statistics need to be determined, $\mathcal{R}$ is the set of all non-identity rules and $\mathcal{I}$ is the set of identity rules. Given the initial plan, the optimizer is invoked to generate the plan space (lines 1–3) and the set of possible SEs over all the plans ($\mathcal{E}$). Since we are interested in estimating the cardinalities of all SEs, we add the cardinality statistic to the $tobecomputed$ set (lines 4–5). Lines 6–16 iterate over all the statistics in the $tobecomputed$ set. For each statistic s_e to be computed, the plans generated for that SE are looked up from the $\mathcal{P}$. For each plan, the rules matching from $\mathcal{R}$ are determined and each such rule is applied to generate a set of statistics (S) that can be used to compute s_e (lines 7–11). If any statistic (s', e') in S is not already considered, it is added to the $tobecomputed$ set (lines 12–14). The set S is also added as a CSS for s_e in the output (line 15). Further, s_e is added to the $computed$ set once all the plans for e have been considered (line 16). At the end of the run, the output C has the CSS for all the s_e that are relevant. The final step is to apply the identity rules to generate additional CSS (lines 17–21). A check is made to ensure that no new statistics are generated in this step. This is because, the identity rules can lead to a blowup in the number of statistics. For example, by repeatedly applying I2, we can see that H^a_T can be computed from a histogram on any subset of attributes containing a, which can be exponential in the number of attributes. However, it is always cheaper to maintain a histogram on a smaller set of attributes. Thus, we should not generate histograms on more attributes, unless they have been already generated by some other rule. This will be explained further through an example below.

4.3 Example

Consider the plan space for a flow as shown in Figure 6. Figure 6(a) is the original plan and Figure 6(b) is the alternative plan generated by the optimizer. The SEs for this plan space are $(E) =$

Alg. 1 ***Generate CSS for a ETL workflow***

Input. $Workflow\ W$
Input. $Ruleset\ \mathcal{R}, IdentityRuleset\ \mathcal{I}$
Output. $C = \{(s_e, \{s'_{e^{\square}}\})\}\ the\ CSSs\ for\ required\ statistics$

```
1.  C = ∅; tobecomputed = ∅; computed = ∅
2.  Invoke optimizer on W to generate plan space P
3.  E = {e : (e, P_e) ∈ P}
4.  for (e ∈ E)
5.      Add (|e|, e) to tobecomputed
6.  while (tobecomputed ≠ ∅)
7.      (s, e) = tobecomputed.pop()
8.      P_e = P[e]
9.      for (p ∈ P_e)
10.         for (r ∈ R matching p.op and s)
11.             S = r(p, s)
12.             for ((s', e') in S)
13.                 if ((s', e') ∉ computed)
14.                     add (s', e') to tobecomputed
15.             add (s_e, S) to C
16.     add (s, e) to computed
17. for ((s_e, S) in C)
18.     for (r in I)
19.         apply r to(s', e') ∈ S to get S'
20.             if (S' contains only statistics
                    already generated)
21.                 add(s_e, S') to C
```

$\{O, P, C, O \bowtie P, O \bowtie C, O \bowtie P \bowtie C\}$. Note that the plan joining C with P is not generated since it is a cross product. For brevity, we denote $A \bowtie B$ as AB. The algorithm starts with $tobecomputed = \{|O|, |P|, |C|, |OP|, |OC|, |OPC|\}$. The plan space for SE OPC is $\{OP \bowtie C, OC \bowtie P\}$. Let us consider the plan $OP \bowtie C$. By applying J1, we get the CSS $\{H^{cid}_{OP}, H^{cid}_{C}\}$ for $|OPC|$. H^{cid}_{OP} and H^{cid}_{C} are added to $tobecomputed$ since they are required by this CSS. Similarly, the plan $OC \bowtie P$ would produce the CSS $\{H^{pid}_{OC}, H^{pid}_{P}\}$. Next, we consider the statistic H^{pid}_{OC} in $tobecomputed$, which has a single plan $O \bowtie C$. Applying, rule J2 we get the CSS $\{H^{pid,cid}_{O}, H^{cid}_{C}\}$. Rule J5 could also apply in this case, which would generate the CSS $\{H^{pid,pid}_{OPC}, H^{pid}_{P}, H^{pid}_{\overline{O}^{pid}_{\bowtie P}}\}$. Similarly, other all the statistics from $tobecomputed$ are processed to generate the CSS for each of them. Now consider the CSS $\{H^{cid}_{OP}, H^{cid}_{C}\}$ for $|OPC|$. If we apply rule I2 to H^{cid}_{OP} we get $H^{cid,pid}_{OP}$ which is a statistic already generated by other rules. Thus we consider $\{H^{cid,pid}_{OP}, H^{cid}_{C}\}$ as another CSS for $|OPC|$. However, $H^{cid,X}_{OP}$, where X is some other attribute of OP is not considered since $H^{cid,X}_{OP}$, doesn't get generated by any of the regular rules. Finally, note that we restrict the number of CSS by applying the rules to only one level in a plan and not recursively. For example, consider the CSS $\{H^{cid}_{OP}, H^{cid}_{C}\}$ generated for $|OPC|$. If we expand this recursively, we would also get the CSS $\{H^{cid,pid}_{O}$, H^{pid}_{P}, $H^{cid}_{C}\}$ for $|OPC|$. However, we do not do so, since the CSS $\{H^{cid,pid}_{O}$, $H^{pid}_{P}\}$ will be generated for H^{cid}_{OP}, which will cover this option.

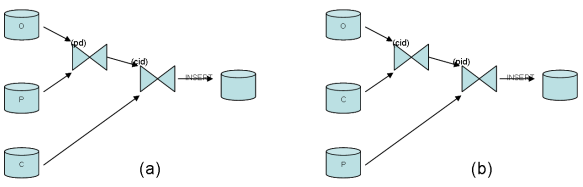

Figure 6: Plan space for generating CSS

5. OPTIMIZATION FRAMEWORK

As seen in Section 4, there are multiple possible CSS for each statistic of a SE. The cost of observing the statistics in a CSS could vary widely. There could be multiple cost metrics such as the CPU-cost of observing the statistics, the memory overhead for maintaining the statistics, etc. The goal of this step is to select a optimal set of statistics with respect to the cost metric such that at least one CSS for the cardinality of each SE in $\mathcal{E}$ is covered. As described earlier, this coverage ensures that the cost of any possible plan can be estimated.

The simple approach of choosing the least cost CSS for each SE is not globally optimal, since there is an amortization of the cost of statistics that are common across the CSS. For example, in Figure 5, assume that the costs of the CSSs for the cardinality of $T_{1,2}$ and $T_{1,3}$ are as depicted in table in fig 7.

s_e	CSS	Cost
$\|T_{1,3}\|$	**1**. $\|T_{1,3}\|$	9
	2. $H^{J_{13}}_{T_1}$, $H^{J_{13}}_{T_3}$	$9+1=10$
$\|T_{1,2}\|$	**3**. $H^{J_{12}}_{T_1}$, $H^{J_{12}}_{T_2}$	$9+3=12$
	4. $H^{J_{13}}_{T_{123}}, H^{J_{13}}_{T_3}, H^{J_{12}}_{\overline{T}^{J_{13}}_{1}}, H^{J_{12}}_{T_2}$	11

Figure 7: Cost of statistics

If we choose the least cost CSS for each statistic, the choices are CSS-1 for $|T_{1,3}|$ and CSS-4 for $|T_{1,2}|$. The complete set of statistics is $\{|T_{1,3}|, H^{J_{13}}_{T_{123}}, H^{J_{13}}_{T_3}, H^{J_{12}}_{\overline{T}^{J_{13}}_{1}}, H^{J_{12}}_{T_2}\}$, leading to a total cost of $9+11=20$. However, if $J_{13} = J_{12}$, i.e. T_1 joins with T_2 and T_3 on the same attribute, it can be seen that $H^{J_{12}}_{T_1}$ is the same distribution as $H^{J_{13}}_{T_1}$. Thus if we had chosen CSS-2 and CSS-3 to cover the two statistics, the complete set of statistics is $\{H^{J_{12}}_{T_1}, H^{J_{12}}_{T_3}, H^{J_{12}}_{T_2}\}$, leading to a total cost of $9+1+3=13$, which is less than the cost of choosing CSS-1 and CSS-4. This is due to the fact that the cost of $H^{J_{12}}_{T_1}$ is shared between CSS-2 and CSS-3.

5.1 Problem Formulation

Let $\mathcal{S} = \{s_1, s_2, \ldots\}$ be the set of all statistics over all the SEs. Depending on the initial plan, some of these statistics are directly observable. Let $\mathcal{S}_\mathcal{O} \subseteq \mathcal{S}$ denote the set of observable statistics. Further, let $\mathcal{S}_\mathcal{C} = \{|e| : e \in \mathcal{E}\}$ denote the set of cardinality statistics over all the SEs. Each $s_i \in \mathcal{S}$ has a set of CSSs generated by the rules. Let CSS_{ij} denote the jth CSS for s_i. Each CSS specifies a set of statistics that can be used to compute s_i. The problem can be defined as finding a set of statistics to actually observe $\mathcal{S}'_\mathcal{O} \subseteq \mathcal{S}_\mathcal{O}$ such that it satisfies two properties:

1. Each $s \in \mathcal{S}_\mathcal{C}$ is computable. A statistic s is computable if either it is directly observed, i.e. $s \in \mathcal{S}'_\mathcal{O}$ or at least one of its CSS is covered. A CSS is covered if all the statistics in it are computable.

2. $\mathcal{S}'_\mathcal{O}$ is optimal in terms of the cost of observation.

Therefore, in principle the problem is to find a subset of $\mathcal{S}_\mathcal{O}$ under some constraints. This can be modeled as an extended version of the classical *Hitting-Set Problem* [13] in literature. This problem is known to be NP hard and can be solved using a linear programming formulation.

5.2 LP Formulation of the problem

We use a $0-1$ integer linear program formulation. A variable x is associated with $\mathcal{S}_\mathcal{O}$; the value of variable x_i is 1 if the corresponding statistic s_i is being observed. A variable y is associated with $\mathcal{S}$; the value of the variable y_i is 1 if the corresponding statistic s_i is computable. A variable z is associated with the set of CSSs such that, z_{ij} is 1, if the corresponding CSS_{ij} is *covered*.

To ensure that a CSS is declared *covered* only if all the constituent statistics are computable, the following set of constraints are introduced, one for each CSS_{ij}, i.e. $\forall_i \forall_j$.

$$\sum_{k:s_k \in CSS_{ij}} y_k \geq z_{ij}.|CSS_{ij}|$$

A statistic whose only CSS is a trivial CSS is computable if and only if it is directly observed. For each such $s_i \in \mathcal{S}_\mathcal{O}$, the following condition is introduced:

$$y_i = x_i$$

The 'only if' condition is not required for statistics that have a non-trivial CSS, since it can be computable without being observed. For each such $s_i \in \mathcal{S}_\mathcal{O}$, the following condition is introduced:

$$y_i \geq x_i$$

Similarly, to ensure that a statistic is deemed *computable* if and only if any of its CSSs are *covered*, the following set of constraints are introduced, one for each s_i, i.e. $\forall i$.

$$y_i \leq \sum_j z_{ij}$$

This covers the 'only if' condition. For the 'if' condition, the following constraints are introduced, one for each j, i.e. $\forall j$.

$$y_i \geq z_{ij}$$

Finally, to ensure that each statistic in $\mathcal{S}_\mathcal{C}$ is computable, the following set of constraints are introduced, one for each $s_i \in \mathcal{S}_\mathcal{C}$, i.e. $\forall_{i:s_i \in \mathcal{S}_\mathcal{C}}$,

$$y_i \geq 1$$

With this formulation, the objective of the LP is to just optimize the following function:

$$\min \sum c_i.x_i$$

The statistics for which $x_i = 1$ are the ones that need to be observed.

Consider the example shown in Figure 5. The set of statistics over all SEs ($\mathcal{S}$), the statistics which are observable ($\mathcal{S}_\mathcal{O}$), the statistics that needs to be computable $\mathcal{S}_\mathcal{C}$ and their respective cost for observation (c_i) are captured as input to LP as shown in Figure 8. The statistic $|T_2|$ i.e. s_2 is observable, needs to be computed and therefore is marked 1 in both the vectors, while $|T_{12}|$ is not directly observale (since subexpression T_{12} is not part of the current plan) but needs to be computed (because some other ordering of the workflow might need it to compute the cost of the plan) and so it is marked 0 and 1 in $\mathcal{S}_\mathcal{O}$ and $\mathcal{S}_\mathcal{C}$ respectively. These along with all the CSS (like those specified in Figure 7) forms the input for the LP and the output is the set of statistics to observe whose total cost is minimal such that $\mathcal{S}_\mathcal{C}$ is covered.

5.3 Greedy Heuristics

The LP formulation could take a long time to solve since $\mathcal{S}$ can be quite large. In such a case, greedy heuristics could be used to arrive at a good solution. One simple heuristic is, in each round, to pick the CSS with the least cost from the set of CSSs that cover at least one of the uncovered statistics in $\mathcal{S}_\mathcal{C}$. After each step, the newly covered statistics are removed from the set of uncovered statistics. Also, the costs of the remaining CSSs are reduced based on the statistics picked in this step, since these statistics would be already available.

5.4 Cost Metrics

The cost of a CSS can be measured in terms of various metrics. We consider two metrics - the memory overhead and the CPU cost. The memory overhead for measuring a histogram on a set of attributes is equal to the number of distinct values of that set of attributes. However, since the exact number of distinct values over the output of a SE may not be known, we conservatively use the number of all possible values. Thus, the memory requirement for a single attribute histogram is proportional to the cardinality of the attribute. For histograms on multiple attributes, the memory required is the product of the cardinalities of the constituent attributes.

The memory overheads of the various statistics are summarized in Table. The CPU cost of measuring the statistic is proportional to the number of tuples in the SE on which the statistic is measured, since for each tuple, the statistic needs to be updated. Thus, to compute the actual CPU cost, we would need the sizes of the SEs, which is what we are trying to estimate using the statistic. We break this circular dependency by using the SE sizes computed from the previous runs. In the first run, we use a coarse approximation based on independence assumptions, since no previous data is available.

Statistic	Memory
$\|T\|$	1
$\|a_T\|$	$\|a\|$
H_T^a	$\|a\|$
$H_T^{a,b}$	$\|a\|\,\|b\|$

6. ENHANCEMENTS

6.1 Optimization under Resource Constraints

Till now, we have assumed that all the resources necessary for identifying the statistics and computing them are available. However, this assumption may not hold in some cases. For example, in Figure 1, to compute the cardinality of IR $Orders \bowtie Customer$, a CSS is $\{H_{Orders}^{Customer\_id}, H_{Customer}^{Customer\_id}\}$. The memory required for this CSS is $2 * |Customer\_id|$ (from Table in Section 5.4), which may not be available.

Generally the cost of the trivial CSS is far less (only 1 counter) than the cost of other CSSs. Therefore, if resources are constrained, one extreme approach is to estimate the cardinality of each SE using its trivial CSS. However, all SEs are not observable in the given initial plan. The initial plan could be altered in a subsequent run so that the statistic that is not observable in the current run could be observed in the reordered plan. In the current example, the trivial CSS would require observing the cardinality of $Orders \bowtie Customer$. However, this trivial CSS is not observable in the initial plan (Figure 1(a)). To address this, the ETL plan can be re-ordered as shown in Figure 1(b), which makes $|Orders \bowtie Customer|$ directly observable.

Thus, repeated execution with plan re-ordering can be exploited to ensure that the optimal plan for an ETL is computed even under

$\mathcal{S}$	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}	s_{11}	s_{12}	...
	$\|T_1\|$	$\|T_2\|$	$\|T_3\|$	$\|T_{12}\|$	$\|T_{13}\|$	$\|T_{23}\|$	$\|T_{123}\|$	$H_{T_1}^{J_{12}}$	$H_{T_2}^{J_{12}}$	$H_{T_3}^{J_{13}}$	$H_{T_{123}}^{J_{13}}$	$H_{\overline{T}_1}^{J_{12}}$	...
$\mathcal{S}_\mathcal{O}$	1	1	1	0	1	0	1	1	1	1	1	1	...
$\mathcal{S}_\mathcal{C}$	1	1	1	1	1	1	1	0	0	0	0	0	...
c_i	1	1	1	∞	1	∞	1	100	100	100	10	30	...

Figure 8: Example LP Formulation

constrained resources. This is the approach followed in [6, 8]. Our method is a natural generalization of these approaches, in which we don't restrict only to the trivial CSS. We use a mix of trivial CSSs and other CSSs, depending on the available memory, thus reducing the number of plan re-orderings.

6.2 Integrating existing statistics from source systems

Sometimes, a few statistics may already be available, especially when the source systems are relational DB systems. The optimal statistics identification framework can be easily extended to take advantage of these statistics. All the statistics that are available can be added by default to the set of observable statistics $\mathcal{S}_\mathcal{O}$ and their costs c_i set to 0. This ensures that the framework will always pick these statistics to cover as many of the statistics in $\mathcal{S}_\mathcal{C}$ as possible.

7. EXPERIMENTAL EVALUATIONS

This section details some experiments to evaluate the effectiveness of statistics identification framework proposed in the previous sections. We are intentionally focusing on establishing the effectiveness of the statistics gathering and determination of optimal statistics to observe, rather than the effectiveness of the optimization itself. The previous works in the literature touch upon that and establish the necessity for cost-based optimization of ETL workflows. Our focus here is to determine the optimal statistics to observe, so that subsequently cost-based optimization of the workflow is possible.

The set of workflows used for the experiments were a representative set of 30 workflows, motivated from a draft version of TPC-DI benchmark being prepared for benchmarking ETL workflows. All the workflows were designed in IBM InfoSphere DataStage V9.1 [10]. Our *Optimal Statistics Identification* module is not integrated with any ETL designer component like DataStage. Therefore all the workflows were exported as XMLs from Datastage to be consumed by our module. we simulated a simple join-order purmutation generator which takes the optimization blocks and UDF boundaries into consideration, to generate different sub-expressions.

All the experiments were run on a Intel Core i5 machine with 2.6 GHz CPU and having 4GB of RAM. The machine configuration will effect only the timings of the run, while the others stay invariant.

The data characteristics of the input relations like table cardinalities, unique values of an attribute (note that we dont need the actual data) are synthetically generated and are as shown in the adjacent table. These are generated from Zipfian distribution with a high skew.

Stat	Card	UV
Max	417874	417874
Min	3342	102
Mean	104466	65768
Median	52234	6529

7.1 Complexity of Workflows

Figure 9 gives an idea of the complexity of the workflows we have used for the experiments. The graph captures the number of SEs, the number of CSS formed without and with the union-division method. From the design perspective, the ETLs range from simple linear ETLs having only one execution plan to complex ETLs having 8-way joins and many transformations. The graph captures that by comparing the number of SEs and the CSSs. The higher the number(both the SEs and CSS), the complex the ETL is.

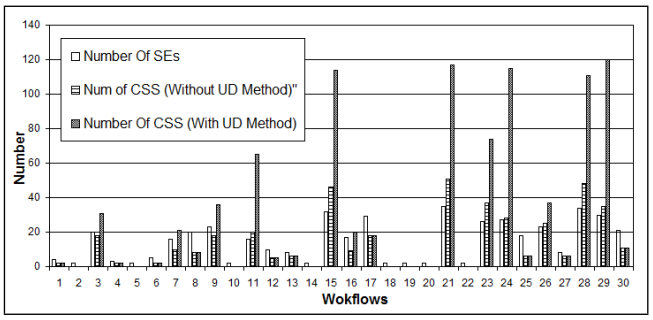

Figure 9: Complexity of the Workflows

For example, workflow 21 is a complex workflow having multiple transformations and a 8-input join. We can also see that the union-division method introduces quite a few additional CSS to choose from. Of course the additional CSS introduced increased the search space of the optimal statistics to choose from and so we have measured the additional time overhead that the additional CSS introduce.

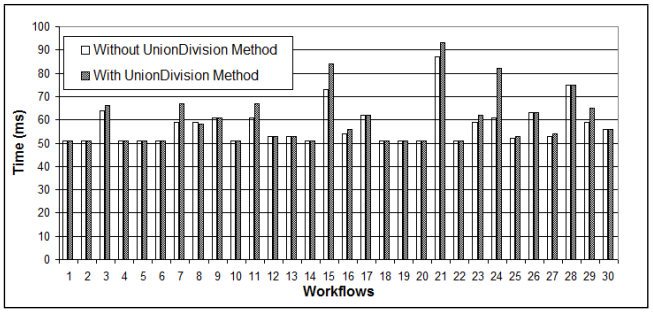

Figure 10: Time Taken for Statistics Identification

Figure 10 captures the time taken for CSS generation using different rules as specified in the previous sections and also the time taken by the LP solver for identifying the optimal statistics to observe. The total time required for identifying the optimal statistics is within 100ms for all the workflows, which is quite reasonable considering that this is an offline process. Further, it can be seen that the additional CSS generated by the union-division method doesn't add any considerable overhead.

7.2 Memory Overhead

Figure 11 shows the memory required by each of the workflows to observe the optimal statistics determined using the previously discussed LP formulation. We can see from the figure that with some additional memory, all the statistics necessary for determining the cost of any re-ordering of the plan can be computed. The units for memory is an abstract unit representing the number of integers (using coarse approximation as explained in Section 5) to be stored. For example, for workflow 16, we need approximately 70000 units of memory and if an integer takes about 4 bytes, then the memory required is about 273KB.

It can be seen from the figure that there are few instances where the new union-division method introduced new choices of CSS, which reduces the amount of memory required. For example, in case of workflow 3, the amount of memory required with and without our method is 29922 and 1811197 units respectively. Of course for few workflows, the CSS generated by union-division method was not optimal and so was not chosen. For example, for workflow 23, the memory required when union-division method is not employed was 3444 units. while for the same workflow, the CSS generated using union-division method was almost twice as costly at 6951 units. But since we are selecting the optimal CSS, the first one is chosen.

For some of the workflows the amount of memory required is high and possibly could be more than the allowed memory limit. In those cases, it could be possible to observe a subset of the CSS that fit with in the memory limit in the current run and re-execute the workflow with a re-orderded plan(s), which allows observation of the rest of the statistics. Developing the techniques for determining the optimal statistics with plan re-ordering is part of our plan for future extensions to the current work.

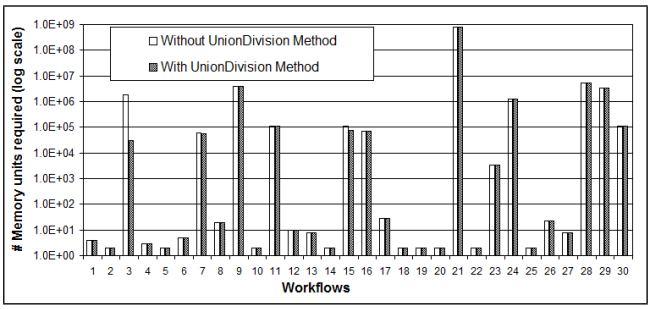

Figure 11: Memory required for observing optimal CSS

7.3 Comparison with existing methods

In this section, we compare with existing techniques such as [6] that observe only trivial CSSs and use plan re-orderings to cover all the SEs. Observing the trivial CSS corresponds to observing only the cardinalities (no distributions) at various points in the plan. This is a quick, easy-to-implement and low-overhead method of passive monitoring [20] that can be used to get the actual cardinalities of SEs which are part of the plan being executed. However, the trivial CSS of all the SEs may not be observable in a single plan. This can be handled by repeating the query execution with different plans such that each SE is covered in some plan. This approach of repeated execution with plan modification was described in [6], in which they determine the cardinality of all the SEs just by observing the cardinalities in the previous runs and then use them for optimizing the subsequent runs of the query. We first derive a formula for the lower bound on the number of re-orderings:

For a workflow, which has a 5-relation join all the 5 SE which cover the base relations and the SE representing the final output are covered by any plan. Therefore, the number of SEs that actually need to be covered are $31 - (5 + 1)$. Also any plan for this query contains 9 SE, out of which 6 are the base relations and the final output. Therefore the number of actual SEs that can be *covered* by a single plan is 3. That is in general for a 'n' table join flow, the number of SEs that actually have to be *covered* is $2^n - (n + 2)$. While, the number of SEs that can be observed (and therefore *covered*) for a given plan is $(n - 2)$. Therefore, the minimum number of executions necessary to cover all SEs without considering any semantics of the query is: $\left\lceil \frac{2^n-(n+2)}{(n-2)} \right\rceil$. Therefore, for the above example, we need 9 executions. Clearly, semantics of the query like cartesian product and the metadata information like whether the join are simple dimension-lookups etc., can be exploited to reduce the number of SEs that need to be covered and thus the number of executions necessary.

If only trivial CSSs are considered, then the number of executions necessary to cover all SEs for our experimental workflows are shown in Figure 12. For each workflow, we worked out one possible solution of plan re-orderings that would cover all the SEs. This gives an upper bound on the number of re-orderings that are necessary. We also plot *Min executions*, which is the lower bound computed using the formula above.

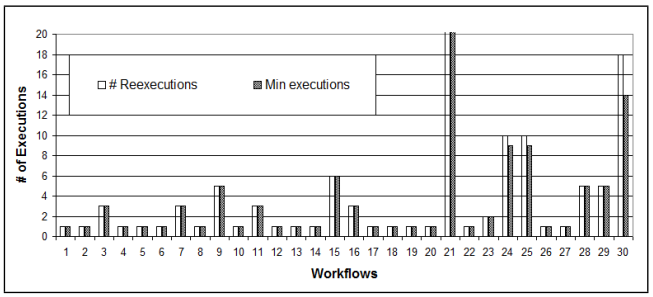

Figure 12: Number Executions to cover all SEs

From the figure, it can be seen that to cover all the different SEs for workflow 30 we need at the minimum 14 executions, while we could find a re-ordering which we required 18 executions. Note that there could be a better re-ordering which could require still less executions, but certainly not less than 14.

The reason for the number of executions to be 1 for quite a few ETLs is because either they are very simple linear ETLs with just one execution plan or have joins across optimization block boundaries. As discussed in the earlier sections, operators across the optimization blocks doesn't commute and therefore even those plans ended up having single execution plan.

Also, In few cases it could be possible that the number of executions necessary to cover all SE can be large. For example, 14 executions necessary in the case of workflow 30, means that the optimal plan for the query can be found only after so many number of executions, which might not be acceptable. Using the techniques in this paper, all the SEs can be covered using the initial run itself, if sufficient memory is available. Even if there is a memory limit, it could be possible to reduce the number of executions, if CSSs are observable with in that memory limit across multiple executions.

For workflow 21 in the benchmark, the *Min executions* necessary was 41, while the the one we found required > 70 executions to cover all the CSS.

8. FUTURE WORK

8.1 Modeling errors

When generating CSSs, different types of statistics are explored and the current work assumes that all these statistics are always accurate. Generally frequency histograms are bucketized for a range of values, and thus the selectivity estimates computed using them introduce error. Section 3 briefly discusses this. When all the statistics are accurate, all of them can compete for optimality equally. But if each of the statistic has an associated error in estimation, then the optimization function needs to consider even the *allowed error* along with the *memory constraints*.

8.2 Space-time tradeoff with errors

Given a workflow, its optimal plan could be determined using the initial plan, given there is enough memory to observe and store all the necessary statistics. But it is generally not the case. So there is a trade-off between the amount of memory allowed and the number of (re)executions. When the statistics are not accurate but introduce errors, this space-time tradeoff might have to be extended to accommodate the *allowed error*.

9. CONCLUSIONS

In this paper, we addressed the problem of cost based optimization of ETL workflows in the case where the statistics on the input relations are either missing or incomplete. We proposed a new framework that identifies all necessary statistics by using a formulation for determining optimal statistics that can enable cost-based optimization of the given query. Also new techniques for determining the cardinality of operators are proposed. The use of metadata, cross-product rules and rules for cardinality estimation drastically reduces the statistics that are needed to estimate all the cardinalities. Experimental results on the ETL benchmarks show that with a small memory overhead, it is possible to measure all the statistics needed in a single execution of the plan.

10. ACKNOWLEDGMENTS

We thank Manish Bhide from IBM and Ravindra Guravannaver from IIT Hyderabad for their valuable comments during our discussions. Thanks to Vinayaka Pandit and Krishnasuri Narayanam for their help in formulating and modelling LP. We also thank anonymous reviewers for their thorough feedback.

11. REFERENCES

[1] A. Aboulnaga and S. Chaudhuri. Self-tuning Histograms: Building Histograms Without Looking at Data. In *SIGMOD Conference*, pages 181–192, 1999.

[2] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *SIGMOD Conference*, pages 261–272, 2000.

[3] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *SIGMOD Conference*, pages 263–274, 2002.

[4] N. Bruno and S. Chaudhuri. Conditional selectivity for statistics on query expressions. In *SIGMOD Conference*, pages 311–322, 2004.

[5] S. Chaudhuri, V. R. Narasayya, and R. Ramamurthy. Diagnosing Estimation Errors in Page Counts Using Execution Feedback. In *ICDE*, pages 1013–1022, 2008.

[6] S. Chaudhuri, V. R. Narasayya, and R. Ramamurthy. A pay-as-you-go framework for query execution feedback. *PVLDB*, 1(1):1141–1152, 2008.

[7] C.-M. Chen and N. Roussopoulos. Adaptive selectivity estimation using query feedback. In *SIGMOD Conference*, pages 161–172, 1994.

[8] H. Herodotou and S. Babu. XPLUS: A SQL-Tuning-Aware Query Optimizer. *PVLDB*, 3(1):1149–1160, 2010.

[9] IBM. IBM Infosphere Datastage Balanced Optimization. IBM InfoSphere DataStage and InfoSphere QualityStage, Version 8.5 Documentation, Dec. 2011.

[10] IBM. IBM InfoSphere DataStage and InfoSphere QualityStage, Version 9.1 Documentation, 2013.

[11] Informatica. How to Achieve Flexible, Cost-effective Scalability and Performance through Pushdown Processing. Whitepaper, Nov. 2007.

[12] N. Kabra and D. J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In L. M. Haas and A. Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 106–117. ACM Press, 1998.

[13] R. Karp. Reducibility Among Combinatorial Problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[14] N. Kumar and P. S. Kumar. An Efficient Heuristic for Logical Optimization of ETL Workflows. In *BIRTE*, pages 68–83, 2010.

[15] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust Query Processing through Progressive Optimization. In *SIGMOD Conference*, pages 659–670, 2004.

[16] A. Simitsis. Mapping conceptual to logical models for ETL processes. In *DOLAP*, pages 67–76, 2005.

[17] A. Simitsis, P. Vassiliadis, and T. K. Sellis. State-Space Optimization of ETL workflows. *IEEE Trans. Knowl. Data Eng.*, 17(10):1404–1419, 2005.

[18] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. QoX-driven ETL design: reducing the cost of ETL consulting engagements. In *SIGMOD Conference*, pages 953–960, 2009.

[19] A. Simitsis, K. Wilkinson, U. Dayal, and M. Castellanos. Optimizing ETL workflows for fault-tolerance. In *ICDE*, pages 385–396, 2010.

[20] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, pages 19–28, 2001.

[21] V. Tziovara, P. Vassiliadis, and A. Simitsis. Deciding the physical implementation of ETL workflows. In *DOLAP*, pages 49–56, 2007.

[22] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost Based Query Scrambling for Initial Delays. In L. M. Haas and A. Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 130–141. ACM Press, 1998.

[23] P. Vassiliadis, A. Simitsis, and S. Skiadopoulos. Conceptual modeling for ETL processes. In *DOLAP*, pages 14–21, 2002.