Efficient Exploitation of Similar Subexpressions for Query Processing

Jingren Zhou
Microsoft Research
jrzhou@microsoft.com

Per-Ake Larson
Microsoft Research
palarson@microsoft.com

Johann-Christoph Freytag
Humboldt-Univ. zu Berlin
freytag@informatik.hu-berlin.de

Wolfgang Lehner
Dresden Univ. of Tech.
lehner@inf.tu-dresden.de

ABSTRACT

Complex queries often contain common or similar subexpressions, either within a single query or among multiple queries submitted as a batch. If so, query execution time can be improved by evaluating a common subexpression once and reusing the result in multiple places. However, current query optimizers do not recognize and exploit similar subexpressions, even within the same query.

We present an efficient, scalable, and principled solution to this long-standing optimization problem. We introduce a light-weight and effective mechanism to detect potential sharing opportunities among expressions. Candidate covering subexpressions are constructed and optimization is resumed to determine which, if any, such subexpressions to include in the final query plan. The chosen subexpression(s) are computed only once and the results are reused to answer other parts of queries. Our solution automatically applies to optimization of query batches, nested queries, and maintenance of multiple materialized views. It is the first *comprehensive* solution covering all aspects of the problem: detection, construction, and cost-based optimization. Experiments on Microsoft SQL Server show significant performance improvements with minimal overhead.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: System—*Query Processing*

General Terms

Algorithms

Keywords

similar subexpressions, query optimization, query processing

1. INTRODUCTION

Database systems frequently encounter queries containing similar subexpressions but today's systems do not automatically exploit such commonalities to speed up query processing. Similar subexpressions may occur in a batch of related queries or within a single complex query with multiple nested subqueries. If a database contains multiple materialized views with similar parts, view maintenance may also produce queries with common or similar subexpressions.

Historically, this problem has been referred to as *multi-query optimization* but multi-query optimization is just one instance of the problem. Current query optimizers optimize queries one at a time and do not identify any commonality in queries. Because they make locally optimal choices for each query, they may miss globally optimal plans.

SQL provides two mechanisms for users to define sharable subexpressions: (virtual) views and common table expressions using the `WITH` clause. A view or common table expression referenced more than once in a query represents a sharing opportunity. However, simply materializing and sharing user-defined expressions is not necessarily the best choice. There may be other sharable expressions that improve performance more. It should be the responsibility of the query optimizer to detect sharing opportunities automatically and to select the best alternative in a cost-based fashion. The following example illustrates the opportunities and the optimization challenges.

Example 1 Consider the following batch of three queries against the TPC-H database that compute summary information for nations and regions.

Q_1:
```
select c_nationkey, c_mktsegment,
  sum(l_extendedprice) as le, sum(l_quantity) as lq
from customer, orders, lineitem
where c_custkey = o_custkey and o_orderkey = l_orderkey
and o_orderdate < '1996-07-01'
and c_nationkey > 0 and c_nationkey < 20
group by c_nationkey, c_mktsegment
```

Q_2:
```
select c_nationkey, sum(l_extendedprice) as le,
  sum(l_quantity) as lq
from customer, orders, lineitem
where c_custkey = o_custkey and o_orderkey = l_orderkey
and o_orderdate < '1996-07-01'
and c_nationkey > 5 and c_nationkey < 25
group by c_nationkey
```

Q_3:
```
select n_regionkey, sum(l_extendedprice) as le,
  sum(l_quantity) as lq
from customer, orders, lineitem, nation
where c_custkey = o_custkey and o_orderkey = l_orderkey
and c_nationkey = n_nationkey and o_orderdate < '1996-07-01'
and c_nationkey > 2 and c_nationkey < 24
group by n_regionkey
```

The first two queries join the same three tables *customer*, *orders*, and *lineitem*, but they group on different columns.

SIGMOD'07, June 12–14, 2007, Beijing, China.

The third query looks similar except it joins an additional table *nation* and groups on a column of that table. All three queries have slightly different selection predicates.

A traditional query optimizer would optimize the three queries separately and generate three query execution plans, each one computing the join of *customer*, *orders*, and *lineitem*.

It is obvious that execution time could be reduced by sharing some intermediate results instead of computing the same joins three times. But there are multiple sharing options. One could share the result of joining tables *customer* and *orders*, the result of joining tables *orders* and *lineitem*, possibly with some aggregation, or the result of joining all three tables, possibly also with some aggregation. It is not immediately clear which solution is the best. *

In this paper, we present an efficient, scalable, and principled solution to reducing query processing time by recognizing and exploiting similar SPJG (selection-projection-join-groupby) subexpressions within a query or among a batch of queries. A query batch can either be submitted by a user or automatically generated. For example, data analysis applications frequently require a batch of queries to be executed. Query batches can also come from a set of decision-support queries or from an application generating reports.

After detecting a set of similar subexpressions, we may construct a *C*overing *S*ub*E*xpression (*CSE*) that contains all tuples and columns required by all the subexpressions. The optimizer evaluates different *CSEs* and determines which ones, if any, to use in the final optimal plan. The chosen *CSE*(*s*) are computed only once and the results are reused to compute other parts of queries.

Our solution has been prototyped in Microsoft SQL Server. Extensive experimental results show very significant reduction in execution time with only moderate increase in optimization time. Our main contributions are as follows.

(a) We introduce a new light-weight mechanism, called *table signatures*, for rapidly finding groups of potentially sharable SPJG subexpressions. The overhead is minimal if there are no sharable expressions.

(b) Our algorithm is the first to consider all detected sharing opportunities and select among them in a cost-based manner. We even allow *CSEs* themselves to share smaller subexpressions. Previous work missed many optimization opportunities.

(c) Our algorithm is the first to correctly optimize queries in the presence of multiple *CSEs* and it fits seamlessly into a commercial-grade optimizer.

(d) Finally, our solution is automatically applied to all query expressions regardless of whether they originate from a single query, a query batch, or view maintenance.

The rest of this paper is organized as follows. We first give an overview of our solution in Section 2. We describe our light-weight mechanism for detecting sharable subexpressions in Section 3. In Section 4, we present how to construct candidate *CSEs* covering a set of similar subexpressions and describe cost-based heuristics to prune out clearly poor choices. We extend optimization to consider multiple *CSEs* in Section 5. We outline three potential applications and present experimental results in Section 6. Finally, we survey related work in Section 7 and conclude in Section 8.

2. SYSTEM ARCHITECTURE

We start with describing our overall design. To assist the reader in understanding the optimization process, we first give a brief overview of transformation-based optimizers built on the Volcano [7] or the Cascades [6] framework.

2.1 Brief Optimizer Overview

Conceptually, an optimizer generates all possible rewritings of a query expression and chooses the one with the lowest estimated cost. A transformation-based optimizer applies local transformation rules on query subexpressions and may generate a large number of expressions during optimization. Graefe [6] describes a *memo* structure that very compactly stores a set of operator trees by consolidating expressions into a DAG (directed acyclic graph).

Nodes in the memo DAG are called *groups*. Each group is assigned a unique group number and is composed of a set of logically equivalent *group expressions*. A group expression contains a single query operator that references its inputs (children) by group numbers. All group expressions within a group generate the same set of result tuples. A group may be referenced by many different group expressions in other groups. Groups and group expressions are consolidated representations of sets of equivalent expressions (operator trees).

Optimization of a query proceeds in several phases with early phases applying fewer transformation rules than later phases. The decision whether to proceed with the next optimization phase depends on the complexity of the query, the cost of the best plan found so far and the elapsed optimization time. Simple, cheap queries may only go through the first phase.

2.2 Solution Overview

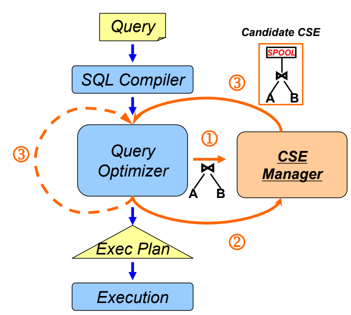


Figure 1: Overall System Architecture

Figure 1 shows the overall system architecture and the three key steps in our solution. The *covering subexpression (CSE) manager* is a new optimizer component. Its function will become clear as we describe our approach in more detail. We also added one more optimization phase, the *covering subexpression (CSE) optimization phase*, which is entered after normal optimization but only if the query is expensive and contains potentially sharable subexpressions.[1]

Step 1: Table signature generation

When a query is submitted, it is first compiled and optimization proceeds in the normal way. The optimizer rewrites the query in different ways by applying transformation rules. For each logically unique expression generated by the optimizer, we compute its *table signature* and register it with the *CSE* manager. This is shown in Figure 1 as *Step 1.* The purpose of this step is to allow detection of potentially sharable expressions with minimal overhead.

[1]A batch of queries is treated as a single complex query by tying them together with a dummy root operator.

A table signature is a very simple abstract of an expression but with the crucial property that *expressions with different table signatures cannot be computed from a covering subexpression*. Table signatures are described in more detail in Section 3. The CSE manager maintains a hash table that records every table signature found in the query with pointers back to the expressions corresponding to the signatures.

If no cheap plan is found during normal optimization, we proceed with *Step 2* and enter the CSE optimization phase.

Step 2: Generation of candidate CSEs

The manager first checks its hash table looking for table signatures that reference two or more expressions originating from different parts of the query. These expressions are the potentially sharable expressions. This check is the first part of *Step 2* in Figure 1. If no such expressions are found, we exit and generate a final execution plan in the normal way.

For each set of potentially sharable expressions, we construct candidate $CSEs$. A candidate CSE is a logical expression with a *spool* operator on top. The spool operator materializes the result in a work table so that it can be reused multiple times. We describe how to construct a CSE covering a given set of expressions and also heuristics to prune out less promising candidates in Section 4.

If at least one candidate CSE is generated, we proceed with *Step 3*, which resumes query optimization to select the best CSE(s) and generate a final execution plan.

Step 3: Optimization with candidate CSEs

We treat each candidate CSE in the same way as a (materialized) view and rely on the optimizer's view matching mechanism to generate equivalent rewrites. The choice of $CSEs$ in the final plan is entirely cost based. If more than one candidate CSE is available, the optimizer may optimize the query multiple times with different sets of candidates. We do not force the optimizer to use $CSEs$ – the optimizer may conclude that the most efficient solution is not to use any $CSEs$ at all. We discuss how to incorporate consideration of $CSEs$ into query optimization in Section 5.

3. TABLE SIGNATURES

Table signatures are at the core of our mechanism for cheaply detecting potentially sharable subexpressions. Because most queries do not contain any similar expression, the mechanism has to be extremely light-weight with minimal overhead during normal optimization.

Definition 3.1 (Table Signatures) *A table signature S_e exists for an expression e iff e represents an SPJG expression. If S_e exists, it is a binary tuple $S_e = [G_e; T_e]$ where*

- *G_e is a boolean indicating whether e contains a group-by operation.*
- *T_e is the set of source tables (or views) in e.*

SPJG signatures described in [2] are similar to table signatures but contain more information and are more expensive to compute. They cannot be used for commonality detection because different instances of the same table have different SPJG signatures.

Table signatures serve as high level abstracts of expressions. Two expressions with different table signatures cannot be covered by the same CSE. Table signatures are used as a fast filter to detect potentially sharable SPJG expressions. For example, $\pi_{c_1,c_2,sum}(\gamma_{c_1,c_2}(\sigma_{p_1}(A) \bowtie \sigma_{p_2}(B)))$ and $\pi_{c_3,min}(\gamma_{c_3}(\sigma_{p_3}(A) \bowtie \sigma_{p_4}(B)))$ have the same table signature $[\mathtt{T}; \{A, B\}]$ even though they have different predicates and column lists. Nevertheless, the two expressions *could* share some computation of join and aggregation. However, neither expression can share computation with $\gamma(\sigma(C) \bowtie \sigma(D))$ which has a different table signature $[\mathtt{F}; \{C, D\}]$.

Operator	Table Signature
Table/View (t)	$S_t = [\mathtt{F}; t]$
Select (σ)	$S_{\sigma(e)} = S_e$, *if* $G_e = \mathtt{F}$
Project (π)	$S_{\pi(e)} = S_e$, *if* $G_e = \mathtt{F}$
Join ($\bowtie$)	$S_{e_1 \bowtie e_2} = [\mathtt{F}; T_{e_1} \cup T_{e_2}]$, *if* $G_{e_1} = G_{e_2} = \mathtt{F}$
Group-by (γ)	$S_{\gamma(e)} = [\mathtt{T}; T_e]$, *if* $G_e = \mathtt{F}$

Figure 2: Rules for Computing Table Signatures (For all other cases not listed, $S_e = \varnothing$)

The table signature for an SPJG expression can be computed efficiently and incrementally by traversing the operator tree in post order and, at each node, applying the rules shown in Figure 2. The output signature is calculated using only the signatures of the input trees and local information. For example, the signature of $\gamma(\sigma(C) \bowtie \sigma(D))$ can be calculated from the signatures of $\sigma(C)$ and $\sigma(D)$ using the join rule.

We store table signatures along with groups and group expressions in the memo to facilitate the optimization process. Table signatures are computed incrementally over group expressions and groups. We omit the details due to space limitation. The overhead of computing signatures is so small that we could not reliably measure it in our experiments.

4. GENERATING CANDIDATE CSES

A set of expressions with the same table signature reference the same input tables so, in principle, it is always possible to create a CSE that covers all the expressions. However, in the worst case, this may require a covering expression consisting of the Cartesian product of the input tables. The result may be so large that it is better to compute each expression from scratch. At the other extreme, we could create a covering expression for every possible subset of expressions and let the optimizer figure out which ones, if any, to use. This is not practical either because it might greatly increase optimization time. The goal of *Step 2* is to generate a small number of *promising CSEs* but without losing opportunities.

4.1 Join Compatible Expressions

To avoid $CSEs$ containing Cartesian products, we require that the covered expressions are *join compatible*, that is, have "enough" joins in common. Virtually all joins are equijoins so we consider only equijoins when defining join compatibility.

Let $E = \sigma_p(T_1 \times T_2 \times \cdots \times T_n)$ be a normalized SPJ expression. The equijoins in E can be summarized compactly by a collection of *equivalence classes* [5] based on the column equality conditions in p. An equivalence class is a set of columns that are guaranteed to be equal in the result of E. Computing the equivalence classes is straightforward and can be found in [5]. From the collection of equivalence classes, we construct the *equijoin graph* for E. The equijoin graph contains one node for each table T_i in E. There is an edge between nodes T_i and T_j if there exists an equivalence class containing a column from T_i and a column from T_j.

Definition 4.1 *Two SPJ expressions E_1 and E_2 over the same set of tables are* join compatible *if the equijoin graph* *constructed from the intersection of their equivalence classes is connected.*

The intersection of equivalence classes C_1 and C_2 is defined in the natural way: for every pair of sets, one from C_1 and one from C_2, output their intersection.

Example 2 Expressions $R \bowtie_{R.a=S.d \wedge R.b=S.e} S$ and $R \bowtie_{R.a=S.d \wedge R.c=S.f} S$ are join-compatible. The intersection of their equivalence classes equals $\{\{R.a, S.d\}, \{R.b, S.e\}\} \cap \{\{R.a, S.d\}, \{R.c, S.f\}\} = \{\{R.a, S.d\}\}$. The corresponding equijoin graph is connected; it consists of two nodes R and S and there is an edge between them (generated by the equivalence class $\{R.a, S.d\}$). However, expressions $R \bowtie_{R.a=S.d \wedge R.b=S.e} S$ and $R \bowtie_{R.c=S.f} S$ are not join compatible. The intersection of their equivalence classes is empty so the equijoin graph has two nodes but no edges. $*$

The simplest way to derive join compatibility among a set of expressions is as follows. For each expression, first extract its full operator tree from the memo and construct its equivalence classes. Then proceed with testing join compatibility by intersecting equivalence classes and checking connectivity of equijoin graphs.

However, it can be somewhat expensive to extract from the memo an operator tree matching a given table signature and to construct its equivalence. It turns out that we can often avoid the extraction step and derive join compatibility for a set of expressions from the join compatibility of their subexpressions. We illustrate the process by an example.

Example 3 Consider the following two expressions

$$e_1 = \sigma_{pr_1}(R) \bowtie_{p_1} S \bowtie_{p_2 \wedge p_3} \sigma_{pt_1}(T)$$
$$e_2 = \sigma_{pr_2}(R) \bowtie_{p_1} \sigma_{ps_2}(S) \bowtie_{p_2} \sigma_{pt_2}(T)$$

All the joins are assumed to be equijoins. If we know that subexpressions $e_1' = \sigma_{pr_1}(R) \bowtie_{p_1} S$ and $e_2' = \sigma_{pr_2}(R) \bowtie_{p_1} \sigma_{ps_2}(S)$ are join compatible, and that subexpressions $e_1^{''} = S \bowtie_{p_2 \wedge p_3} \sigma_{pt_1}(T)$ and $e_2^{''} = \sigma_{ps_2}(S) \bowtie_{p_2} \sigma_{pt_2}(T)$ are join compatible, we can safely conclude that e_1 and e_2 are join compatible.

Here is the reasoning. The equijoin graph of e_1' and e_2' consists of nodes R and S and an edge (R, S). Otherwise the graph would not be connected and the expressions would not be join compatible. Similarly, the equijoin graph for $e_1^{''}$ and $e_2^{''}$ consists of node S and T and an edge (S, T). The equijoin graph of e_1 and e_2 consists of, at least, the union of these two equijoin graphs. The union of the two graphs contains nodes, R, S, and T, and edges (R, S) and (S, T). That is, the union graph covers all tables and is connected. It follows that e_1 and e_2 are join compatible. $*$

However, such optimization may not always work because the optimizer may not have explored enough subexpressions. If so, we fall back on deriving join compatibility by the basic method described in the beginning of this section.

4.2 Covering Subexpressions (CSE)

The join-compatibility analysis divides expressions with the same table signature into join-compatible groups. Each group contains only mutually join-compatible expressions. The next step is to generate candidate $CSEs$ for each group containing more than one expression.

Given a set of target expressions, a covering subexpression (CSE) that contains all tuples and columns required by the target expressions can be constructed as follows. We call the target expressions *potential consumers* of the CSE.

1. Compute equivalence classes for all potential consumers and take their intersection. Create an N-ary join operator, with equijoin predicates from the intersected equivalence classes.
2. Simplify the selection predicate of each potential consumer by deleting any conjunct already included in the join predicate constructed in step 1.
3. Add a covering selection predicate, if any, by OR'ing the simplified predicates. AND the covering predicate to the join predicate constructed in step 1.
4. If aggregation is required, add a group-by operator on top of the N-ary join. Its grouping columns consists of the union of the following: the group-by columns of all potential consumers and all columns referenced in the covering predicate constructed in the step 3. Its aggregation expressions (functions) include aggregation expressions from all potential consumers.
5. If needed, add a projection operator on top. Include as output columns, all columns and (aggregation) expressions that are required to compute the result of a potential consumer.
6. Add a spool operator on top.

Example 4 Consider the following two expressions

$$\gamma_{c_1,c_2}^{e_1}(\sigma_{pa_1}(A) \bowtie_{p_1} B \bowtie_{p_2 \wedge p_3} \sigma_{pc_1}(C))$$
$$\gamma_{c_1}^{e_2}(\sigma_{pa_2}(A) \bowtie_{p_1} \sigma_{pb_2}(B) \bowtie_{p_2} \sigma_{pc_2}(C)).$$

All joins are assumed to be equijoins. Both expressions have table signature $[\mathtt{T}; \{A, B, C\}]$ and are join compatible. We first convert the expression into normal form, which produces $\gamma_{c_1,c_2}^{e_1}\ \sigma_{pa_1 \wedge p_1 \wedge p_2 \wedge p_3 \wedge pc_1}(A \times B \times C)$ and $\gamma_{c_1}^{e_2}\ \sigma_{pa_2 \wedge p_1 \wedge pb_2 \wedge p_2 \wedge pc_2}(A \times B \times C)$. The common join predicates are p_1 and p_2. To create the covering predicate, we first drop p_1 and p_2 from both predicates and then OR the result. We also need a group-by operator. Suppose the covering predicate references columns c_2 and c_3. The group-by columns for the operator are then $\{c_1, c_2\} \cup \{c_1\} \cup \{c_2, c_3\} = \{c_1, c_2, c_3\}$ and the aggregation expressions are e_1 and e_2. The CSE (without the spool operator) is then

$$\gamma_{c_1,c_2,c_3}^{e_1,e_2}\ \sigma_{p_1 \wedge p_2 \wedge ((pa_1 \wedge p_3 \wedge pc_1) \vee (pa_2 \wedge pb_2 \wedge pc_2))}(A \times B \times C)$$

4.3 Candidate Generation

Suppose we have a group with four consumers. What candidate $CSEs$ should we generate? A simple solution would be to create a single CSE that covers all four consumers. However, this is not necessarily the best solution. The CSE may produce a very large result that does not fit any of its consumers well. This may happen, for example, if the consumers require different sets of columns or different sets of tuples. In that case, each consumer may have to "wade through" a lot of data that it does not need. This illustrates the fact that we must consider multiple candidate $CSEs$, each one covering some subset (or all) of the consumers. We cannot *a priori* decide that a single CSE covering all consumers is the best solution.

Ideally, we would create a candidate CSE for every subset of consumers. But this exhaustive algorithm is exponential in the number of consumers. Instead we use the greedy algorithm described in Algorithm 1. We create one trivial CSE for each consumer, which is, of course, exactly the same as its only consumer. We start with a trivial CSE and greedily merge in one other trivial CSE at a time to maximize the merging benefit until no more beneficial merging is available.

Algorithm 1: CreateCandidateCSE(E)

```
Input: ExprSet E  /* Set of join compatible expressions        */
                  /* with the same table signature             */
Output: CandidateSet C
CandidateSet R, M;    Bool IsCandidate;
R = TrivialCandidateSet(E);    C = ∅;
Apply heuristics in Section 4.3.1 and 4.3.2 to reduce R.
while |R| > 1 do
    Pick r ∈ R;
    R = R − {r};    M = E − {r};    IsCandidate = F;
    while M ≠ ∅ do
        Pick m ∈ M which maximizes the benefit Δ;
        /*Δ defined in Section 4.3.3.                          */
        if Δ > 0 then
            r = Merge(r, m);   IsCandidate = T;
            M = M − {m};    R = R − {m};
        else
            break;  /*no more beneficial merging exists        */
        end
    end
    if IsCandidate then C = C + {r};
end
return C;
```

If there are still trivial $CSEs$ that have not been merged, we apply the algorithm again to the remaining trivial $CSEs$.

During generation, several heuristic rules are applied to prune out candidate $CSEs$ that are not promising or are less promising than other candidates. The goal is to reduce the number of candidate $CSEs$ generated but without missing opportunities. We have to be careful to keep the overhead of the heuristic rules low. In particular, we cannot afford to fully cost each candidate CSE, as done in [1].

Fortunately, we are not completely helpless. Because normal optimization phases have completed before entering the CSE phase, the memo structure contains a wealth of information that can be exploited. It contains the best solution found so far for the query and its final cost C_Q. A group may have been optimized several times, each time with different requirements on the solution, for example, unsorted or sorted on a given set of columns. For each group and requirements, we know the best solutions found, if any, and cost bounds, including both the upper bound and the lower bound. Our heuristics exploit these optimal costs or cost bounds to prune out candidates that do not appear promising. The actual costs for using a candidate are computed and evaluated during later optimization as described in Section 5. Our heuristics are applied in a conservative manner in that they are only active if all required cost bound information is available.

In the rest of this section, we describe four important cost-based heuristics. As we shall see in Section 6, they are both effective and efficient. We denote the cost of the best solution found before CSE optimization by C_Q. For a giving candidate, we denote its N potential consumers by $G_1, \ldots, G_N$.

4.3.1 Don't Bother With Cheap Expressions

The first heuristic is based on the observation that only expensive expressions are worth consideration. The criteria is that the total cost of all potential consumers must be a significant part of the overall query cost. Otherwise, the potential improvement is likely to be too small to be worth the potential optimization overhead.

Heuristic 1 *Consider a candidate CSE and denote the* lower cost bound *of its potential consumer G_i by $C^{lower}_{G_i}$. Discard this CSE if there is not enough potential savings, that is, if it satisfies the condition*

$$\frac{\sum C^{lower}_{G_i}}{C_Q} < \alpha$$

where α is a constant. In our experiments, we use $\alpha = 10\%$.

We use the lower cost bound here because it represents either the cost of its optimal solution or the cost of another competing plan. $\sum C^{lower}_{G_i}$ represents the maximum possible contribution from all the consumers. We apply this heuristic both before and after analyzing join compatibility. Applying it before analyzing join compatibility among a set of potential consumers helps discard obviously trivial cases immediately. After join compatibility analysis, we can apply this heuristic again because we may lose some consumers due to join incompatibility and the remaining potential improvement may no longer be compelling.

Example 5 Consider the three queries in Example 1. The join of *customer* and *orders* could be shared by three potential consumers. However, the cost of this join operation is so low, compared to the query overall cost, that we can safely exclude this candidate from further consideration. $*$

4.3.2 Exclude Consumers With Huge Results

Sharing $CSEs$ does not come free of charge. There are three costs associated with a CSE E. First, the expression E is evaluated once. We denote the evaluation cost by C_E. The "spool" operator materializes the result into an internal work table at a writing cost of C_W. Each consumer reads the work table sequentially and performs any required additional computation, such as evaluating compensation predicates, etc. We denote the combined usage cost by C_R. For each candidate CSE, we know what columns it must output to serve all its consumer(s). Together with the estimated cardinality, we can calculate both C_W and C_R based on the estimated data volume written and read.

A candidate CSE that produces a large result has high materialization and reading costs. To avoid generating candidates that produce very large results, we exclude a consumer from consideration if the cost of materializing and reading its result is higher than computing the expression from scratch. We estimate the cost of computing an expression from scratch conservatively by using its upper cost bound, that is, the maximum cost among the optimal plans in the group to which the expression belongs.

Heuristic 2 *Consider a candidate CSE with N consumers. For a given consumer G_i, denote its* upper cost bound *by $C^{upper}_{G_i}$, the cost of materializing its result by C_{W_i}, and the cost of using the result by C_{R_i}. Discard consumer G_i if it satisfies the condition*

$$C^{upper}_{G_i} < C_{R_i} + \frac{C^{upper}_{G_i} + C_{W_i}}{N}$$

Note that if a candidate is created to cover consumer G_i only, the cost of evaluating the candidate is at most $C^{upper}_{G_i}$. In the best case, both the evaluation cost and the materialization cost are shared by all consumers (right side of the inequality). Even so, if it is still cheaper to compute the expression from scratch (left side of the inequality), we can safely exclude this consumer from consideration. The criteria identifies consumer expressions that are cheap to compute but generate a large result.

Example 6 Consider the following two queries that join *customer* and *orders* in a TPC-H database.

Q_4:
```
select *
from customer, orders where c_custkey = o_custkey
```

Q_5:
```
select c_name, c_nationkey, o_totalprice
from customer, orders where c_custkey = o_custkey
```

The join of *customer* and *orders* could be shared between the two queries. However, Q_4 requires all columns from *customer* and *orders* so the cost of just writing the result would be significantly higher than the cost of computing the query from scratch. Therefore, consumer Q_4 should be discarded and no candidate is generated. $*$

4.3.3 Merge Only When Beneficial

Merging two candidates can save redundant computation but it is not always beneficial because the new CSE may produce a larger result than the source CSEs and thus significantly increase the materialization cost and reading cost for its consumers. We should create a merged CSE only when using the merged one is cheaper than using the two source ones separately.

With N final consumers, using a CSE E contributes a total cost of $C_E + C_W + \sum^N C_R$ to the final query cost. As indicated earlier, C_W and C_R can be estimated based on the cardinality of the expression and the set of output columns.

To obtain a correct estimate of the evaluation cost C_E we would have to invoke the optimizer on the merged expression but this may be expensive. Instead we approximate it using the cost bounds of its consumers as follows. Clearly, the cost of the merged expression must be at least as high as the lowest cost bound of each of its consumers. That is, we find the lowest cost bound of each one of its consumers and use the highest among them as a lower cost bound for the merged expression. Denote this lower cost bound as C_E^{lower}.

Heuristic 3 *Consider two candidate CSEs E_i and E_j with N_i and N_j consumers, respectively. $C_{E_i}^{lower}(C_{E_j}^{lower})$ represents the estimated lower bound on evaluation cost, $C_R^i(C_R^j)$ represents the usage cost and $C_W^i(C_W^j)$ represent the writing cost. The expression E created by merging E_i and E_j would have N consumers ($N \leq N_i + N_j$), an estimated lower bound on evaluation cost of C_E^{lower}, a usage cost of C_R and a writing cost of C_W. The benefit for merging is defined as $TotalCost_{E_i} + TotalCost_{E_j} - TotalCost_E$.*

Computing the merged CSE cannot be cheaper than computing one of the source CSEs, that is, $C_E^{lower} \geq max(C_{E_i}^{lower}, C_{E_j}^{lower})$. We estimate benefit Δ as

$$\begin{aligned}\Delta =& (C_W^i + \sum^{N_i} C_R^i + C_W^j + \sum^{N_j} C_R^j) - (C_W + \sum^{N} C_R) \\ & - max(C_{E_i}^{lower}, C_{E_j}^{lower})\end{aligned}$$

Merge the two candidate CSEs only if $\Delta > 0$.

Example 7 Consider the following two queries that join *orders* and *lineitem* in a TPC-H database.

Q_6:
```
select o_orderkey, l_extendedprice
from orders, lineitem
where o_orderkey=l_orderkey and o_orderdate='1995-01-01'
```

Q_7:
```
select o_orderkey, l_extendedprice
from orders, lineitem
where o_orderkey=l_orderkey and o_orderdate>'1995-01-01'
```

Both queries contain a join between *orders* and *lineitem*. We create a (trivial) candidate CSE for each of the two consumers. Is it worthwhile creating a merged CSE E covering both consumers? The result of the merged CSE would be fairly large because consumer Q_7 requires all items ordered after 01/01/1995. On the other hand, before merging, the expression for the other consumer Q_6 is extremely cheap due to an index on *o_orderdate*. Computing Q_6 from the merged CSE would be much more expensive because we would have to scan the whole result of the CSE. In the end, $\Delta = (C_R^{Q_6} + C_W^{Q_6} + C_R^{Q_7} + C_W^{Q_7}) - (\sum^2 C_R^E + C_W^E) - C_{Q_6} < 0$. According to this heuristic, merging is not helpful. $*$

4.3.4 Containment Checking

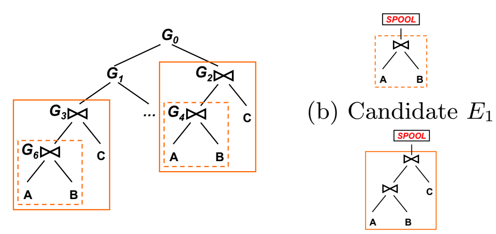
Whenever two or more expressions are equivalent, they also share equivalent subexpressions. This means that whenever we create a candidate CSE for a set of consumers, those consumers may share many subexpressions, for which we could also create candidate $CSEs$. We could blindly create candidate $CSEs$ for all shared subexpressions and let the optimizer determine which ones are the most beneficial. This is wasteful because in many cases we can safely determine that a candidate is dominated by another candidate.

Definition 4.2 (Containment) *A candidate CSE E_c is* contained *by another candidate CSE E_p if*

- *The set of input tables of E_c is a subset of the set of input tables of E_p;*
- *Each of E_c's consumers G_c is a descendant of one of E_p's consumers G_p in the operator tree. That is, in the memo structure, group(G_c) is a descendant group of group(G_p).*

We say that the parent candidate E_p is wider because it references more tables than the child candidate E_c. A wider CSE is usually preferable because it incorporates more shared computation than a narrower one. However, this is not always true; the wider CSE may produce a much larger result such that the narrower one may become more beneficial.

Example 8 Consider the query shown in Figure 3(a). The three-way join of A, B, and C appears twice in the operator tree. We can create a candidate E_2, shown in Figure 3(c), corresponding to the three-way join. At the same time, any two-way join among A, B, and C is also shared by two consumers. For example, candidate E_1 can be created, shown in Figure 3(b), corresponding to the two-way join of A and B.



(b) Candidate E_1

(a) Query Operator Tree (c) Candidate E_2

Figure 3: CSE **Containment**

E_1 has a consumer set of $\{G_6, G_4\}$ and E_2 has a consumer set of $\{G_3, G_2\}$. At first glance, E_2 appears to be preferable because more work is shared. However, if E_2 is much larger than E_1, the materialization and reading costs may be significantly higher than for E_1. If so, the narrower expression E_1 may be preferable. $*$

In this example, the two $CSEs$ have the same number of potential consumers but this is not always the case. As shown later in the experiments, a narrower CSE E_c may have more consumers than a wider CSE E_p. In that case, by definition, E_c is not contained by E_p.

Heuristic 4 *Consider a candidate CSE E_c that is contained by another candidate CSE E_p. Suppose E_c produces a result with estimated size S_c and E_p produces a result with estimated size S_p. Discard the contained E_c if*

$$S_c > \beta \times S_p$$

where β is a constant. In our experiments, we use $\beta = 90\%$.

CSE containment is very common for queries with joins of multiple tables. This heuristic prunes out a large number of small, less promising candidate $CSEs$ and reduces optimization time dramatically.

Example 9 We consider the three queries in Example 1 again and two candidate $CSEs$: E_1 that joins *customer*, *orders*, and *lineitem* and E_2 that aggregates on columns ($c\_nationkey, c\_mktsegment$) after the join. Considered in isolation, each expression seems to be useful and is not pruned out by earlier heuristics. However, E_1 is contained by E_2 and E_2 is even smaller than E_1 because of the aggregation. By the containment heuristic, E_2 is always preferable to E_1. We can exclude E_1 without missing opportunities. $*$

5. OPTIMIZATION WITH CSES

If any candidate CSE remains after the heuristic pruning, we resume query optimization to determine which candidates, if any, to make use of in the final plan. In general, adding another phase to optimization is much cheaper than optimizing from scratch because only some parts of the query are reoptimized and all optimization information gathered in previous phases helps avoid redundant work.

Due to space limitations, we only list important optimization strategies in this paper. We also omit the details on handling stacked $CSEs$, described in Section 5.5.

5.1 General Procedure

We treat candidate $CSEs$ in the same way as materialized views and rely on the view matching algorithms [5] to generate substitute expressions. For example, a substitute may include a compensation predicate over a CSE. The optimizer compares the plan using the CSE against other alternatives in its normal cost-based fashion. The final plan may or may not use the CSE.

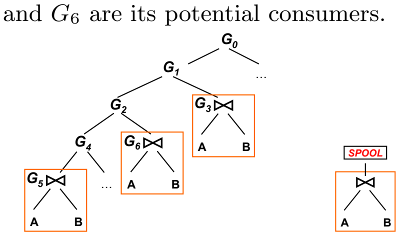
For each candidate, we know exactly which expressions are potential consumers. To avoid reapplying view matching for every expression in the query, we enable the view matching rule only for consumer expressions.

5.2 CSE Costing

Costing CSEs properly is crucial for correct optimization. As discussed earlier, a CSE has a "spool" operator on top, which materializes the result of the expression. There are three costs associated with a CSE. We call the combination of C_E and C_W the initial cost of the CSE.

Normal costing of a spool operator assumes that the operator has a *known* set of consumers. Under this assumption, costing of a spool operator is straightforward. With N consumers, the optimizer splits the initial cost of the spool among all the consumers so that each consumer gets charged a cost of $C_R + \frac{C_E+C_W}{N}$ for using the spool. However, in the case of a CSE, we only know a set of *potential* consumers. There is no guarantee that every consumer will eventually use the CSE. For example, some consumer may choose an even cheaper solution, such as an index operation or using materialized views, etc. Simple cost splitting may result in incorrect costing for the rest of the consumers. We illustrate the issues by an example.

Example 10 Figure 4(a) shows a (simplified) operator tree of a query. G_0 to G_6 indicate which memo groups the operators originates from. The query has three similar subexpressions rooted at G_3, G_5, and G_6. We create a candidate covering all three subexpressions, shown in Figure 4(b). Groups G_3, G_5, and G_6 are its potential consumers.



(a) Memo DAG (b) Candidate

Figure 4: CSE Optimization

The optimizer traverses the operator tree in post order. Consumers G_3, G_5, and G_6 are optimized individually and substitutes using the CSE are generated. When optimizing G_3, the optimizer does not know whether the CSE will also be used by G_5 and G_6. For the reasons described earlier, we cannot split the initial cost into three parts and charge each consumer one third of the cost.

Another possibility is to charge the whole initial cost plus the usage cost for the first consumer and charge only the usage cost for the rest consumers. However, this is not feasible either. It treats the first consumer so unfavorably that its cost of using the CSE is always more expensive that its cost of not using the CSE. As a result, the plan using the CSE would be pruned out by the optimizer and there would be no first consumer. $*$

The correct solution is to charge the usage cost C_R for each consumer that uses a candidate CSE but only add the initial cost once we know the decision of all consumers. But, where and when can we add the initial cost?

Of course, we can always add the initial cost at the root group of the query, that is, group G_0 in Figure 4(a). But this is later than necessary and may waste optimization time. Before reaching the root group, the optimizer may (wrongly) select subplans using the CSE because only usage costs have been charged yet. After adding the initial costs, the final plan may be more expensive than other alternatives. But the optimizer could not tell until it reached the root group G_0. At that time, a lot of optimization work done for the plan turns out to be useless. To avoid this wasted effort, we need to add the initial cost as soon as possible.

Definition 5.1 (Least Common Ancestor) *In an operator tree, the least common ancestor of a set of nodes $\mathcal{S}$ is the lowest node p in the tree such that every node in $\mathcal{S}$ is descendant of p.*

The groups in the memo structure form are connected in a DAG. The least common ancestor for a set of groups $\mathcal{G}$ is the lowest group p in the DAG such that every group in $\mathcal{G}$ is a descendant of p.

The initial cost for a candidate CSE can be safely charged when optimizing the *least common ancestor* group of all its potential consumer groups.

The least common ancestor group for a CSE can be calculated statically before the CSE optimization phase begins. Each group maintains a set of potential consumer groups of a CSE that are its descendants. Information about potential consumers is propagated recursively bottom-up. The first node that has collected information from all consumers of a CSE is the least common ancestor for that CSE. Note that different $CSEs$ may have different least common ancestors.

Query optimization is done by traversing the operator tree in post order. Each subplan produced carries with it information about the plan, including which $CSEs$ it uses and how many times each CSE has been used. At the least common ancestor, two actions are performed.

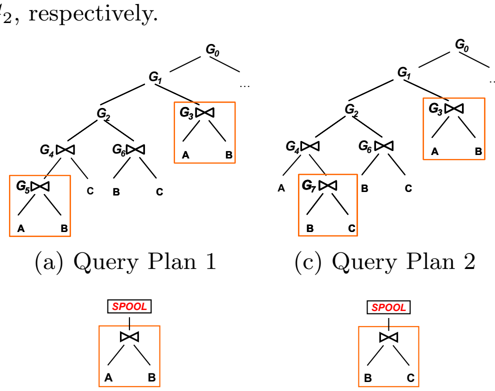
- Discard any plan with only one consumer of the CSE;
- Otherwise, add the initial cost for the CSE.

As a further improvement, we determine the least common ancestor dynamically instead of statically. For example, G_1 is the original least common ancestor for the candidate in Example 10. Suppose the optimizer traverses the tree in the order $G_0 \rightarrow G_1 \rightarrow G_3$. After G_3 has been optimized and if the resulting plan does not use the CSE (possibly due to some cheap index available), all remaining potential consumers come from the left branch of G_1. The optimizer can then dynamically designate G_2 as the least common ancestor for the candidate. By doing so, we can add the initial cost at G_2, possibly pruning expensive plans earlier.

5.3 Multiple Candidates CSEs

So far we have discussed how to extend the optimizer to consider a single candidate CSE. We now consider how to handle multiple candidate $CSEs$. Because the optimizer initially charges only the usage cost for each consumer, it may prematurely prune out useful plans, solely based on the usage costs as illustrated by the following example.

Example 11 Consider the a query with two equivalent (simplified) operator trees extracted from the memo DAG, as shown in Figure 5(a) and (c). The only difference between the two operator trees is the difference in join order of the trees rooted at G_4. Both groups G_5 and G_7 are children of G_4 in the memo. We have two candidate $CSEs$, shown in Figures 5(b) and (d). Their least common ancestors are G_1 and G_2, respectively.



(a) Query Plan 1 (c) Query Plan 2

(b) Candidate E_1 (d) Candidate E_2

Figure 5: Optimization with Multiple Candidates

Optimization proceeds bottom-up. For view substitutes at groups G_5 and G_7, only the usage cost of the corresponding candidate is charged. At G_4, the optimizer chooses the cheaper of the two alternative plan trees and discards the other one. Assume that the usage cost of E_1 is far less than that of E_2. Therefore, the optimizer prefers the plan using E_1 and removes the other one.

However, the initial cost of E_1 can be much higher than that of E_2. After taking the initial costs into account, the plan using E_2 may actually be cheaper. But the optimizer cannot determine this until it processes their least common ancestors G_2 and G_1. By that time, the plan using E_2 has already been discarded. In this case, candidates E_1 and E_2 are mutually exclusive. The optimizer prematurely pruned out the useful plan purely by comparing usage costs. $*$

The solution to this problem is to trigger optimization multiple times, each time specifying a different set of candidate $CSEs$ to be considered. The optimizer may use any candidates in the set but is not required to. The set of candidates for the optimizer to consider is treated as part of required properties, which are propagated top-down as the optimizer traverses the memo structure. At the least common ancestor of a candidate, any returned plan with only one consumer is discarded as described before.

In the previous example, we optimize the query with three different sets, $\{E_1, E_2\}$, $\{E_1\}$, and $\{E_2\}$. The optimizer then compares the three resulting plans with each other and also with other plans generated in normal optimization phases and chooses the cheapest one.

Unfortunately, this means that more optimization is required. The naive way is to optimize the query with every possible combination of candidate $CSEs$. With N candidates available, the number of optimizations would be $2^N - 1$. However, exploring the relationship among different candidates can help us reduce reoptimization significantly.

Definition 5.2 (Competing/Independent $CSEs$) *Consider two candidate CSEs E_1 and E_2 and denote the least common ancestor group of all their potential consumers by G_1^{lca} and G_2^{lca}, respectively. If G_1^{lca} is either a descendant or an ancestor group of G_2^{lca}, E_1 and E_2 are said to be* competing *$CSEs$. If E_1 and E_2 are not competing, they are said to be* independent *$CSEs$.*

If E_1 and E_2 are competing $CSEs$, the optimizer at some level may have to choose between a plan using E_1 and a plan using E_2. Because this decision would be based purely on the usage costs, it could potentially result in a suboptimal plan, as described in the previous example. Additional strategies with different sets of candidate $CSEs$ enabled may have to be evaluated. On the other hand, if E_1 and E_2 are independent, their potential consumers are completely unrelated so the decision whether to use E_1 has no effect on the decision whether to use E_2. Such information can be exploited to prevent unnecessary reoptimization work.

Definition 5.3 (Independent CSE Set) *Let $\mathcal{S}$ be a set of candidate CSEs $\{E_1, E_2, \cdots, E_n\}$. If every candidate $E_i \in \mathcal{S}$ is independent of all other candidates in $\mathcal{S}$, $\mathcal{S}$ is an* independent *CSE set.*

Before describing our pruning algorithm, we first present a few important observation. At each optimization step, some set of candidate $CSEs$ is enabled.

Proposition 5.4 *Suppose $\mathcal{S}$ is a set of* independent *candidate $CSEs$. After the query has been optimized with the set $\mathcal{S}$ enabled, we can skip optimization for any set $\mathcal{S}_i$ such that $\mathcal{S}_i \subseteq \mathcal{S}$.*

If $\mathcal{S}$ is a set of independent candidate $CSEs$, the $CSEs$ in $\mathcal{S}$ have totally unrelated sets of potential consumers and the decision whether to use a candidate or not is not affected by any other candidate in $\mathcal{S}$. If the resulting plan uses a CSE E ($E \in \mathcal{S}$), it must be the case that using E is cheaper than not using E. It also follows that if the resulting plan does no use a CSE E ($E \in \mathcal{S}$), it must be the case that using E is more expensive than not using E. In either case, any optimization with a subset of $\mathcal{S}$ enabled is redundant.

Proposition 5.5 *Suppose $\mathcal{S} = \mathcal{T} \cup \mathcal{R}$, $\mathcal{T} \cap \mathcal{R} = \emptyset$, is a set of candidate CSEs such that all the candidates in $\mathcal{T}$ are independent of all other candidates in $\mathcal{S}$. After the query has been optimized with $\mathcal{S}$ enabled, we can skip optimization for any set $\mathcal{S}_i$ such that $\mathcal{S}_i \subset \mathcal{S}$, $\mathcal{S}_i \cap \mathcal{R} = \mathcal{R}$ and $\mathcal{S}_i \cap \mathcal{T} \subset \mathcal{T}$.*

Proposition 5.5 is a fairly straightforward generalization of Proposition 5.4 and the reasoning why we can skip the indicated sets is similar. The final proposition is based on the properties of the returned optimal plan at each step.

Proposition 5.6 *For each optimization, if the returned optimal plan uses a set $\mathcal{S}^u$ of candidate $CSEs$, the returned plan is optimal also if optimizing with only $\mathcal{S}^u$ enabled.*

This property means that we can skip optimization for $\mathcal{S}^u$. At the same time, we can treat the optimization as having been done with $\mathcal{S}^u$ enabled, and apply Proposition 5.5 to eliminate other combinations.

Overall Procedure: We begin by listing all $2^N - 1$ subsets and sort them based on the number of $CSEs$ in the set. We then perform CSE optimization in descending order, at each step enabling a different set of candidates. After each optimization, we apply Propositions 5.5 and 5.6 to eliminate combinations that have not yet been processed. This process continues until there are no more combinations to process. The final plan is the cheapest plan found.

Example 12 Consider a query with four $CSEs$, $\{E_1, E_2, E_3, E_4\}$. We start by optimizing the query with all four $CSEs$ enabled.

Case 1: $\{E_1, E_2, E_3, E_4\}$ is a independent CSE set. No matter what the returned plan is, we are done. The returned best plan is the final optimal plan.

Case 2: E_1 is competing with E_2 and E_3 is competing with E_4 but E_1 and E_2 do not compete with E_3 and E_4. That is, we have two sets, $\{E_1, E_2\}$ and $\{E_3, E_4\}$, each set with consumers unrelated to the consumers of the other set.

If the returned best plan contains only $\{E_1, E_2, E_3\}$, by Proposition 5.5 and 5.6, we can skip combinations of $\{E_1, E_2, E_3\}$ and $\{E_1, E_2\}$.

If the returned plan uses all the four candidates, in principle, we should try $\{E_1\}$ and $\{E_2\}$ for the first set of consumers, and $\{E_3\}$ and $\{E_4\}$ for the second set of consumers. However, our algorithm may still try combination of $\{E_1, E_3, E_4\}$ (and others). It sounds redundant for the second set of consumers, but, in fact, there is little overhead because the optimizer knows that previous solution for the second set is still usable and returns the plan immediately, as described in the next section. Therefore, in the end, we only carry out the reoptimizations that are necessary.

Due to space limitation we cannot enumerate all possible cases but it is clear that the improved reoptimization strategy can save a lot of unnecessary work. $*$

5.4 Exploiting Optimization History

The CSE optimization phase comes after normal optimization phases. Much optimization history has been collected and can be exploited. Even information collected during CSE reoptimization can be helpful for later reoptimizations with different sets of candidates enabled. Reoptimization with a different set of candidates can be much cheaper than a totally new optimization.

First, we only consider reoptimization for groups whose descendants contain potential consumers. Other groups are not affected by $CSEs$ and their solutions can be reused.

Second, we treat the set of enabled $CSEs$ as part of required properties. Previous optimization history on each group is heavily exploited. For example, if at a given group, the existing solution satisfies the new requirement and the new requirement satisfies the previous requirement, we can deduce that the existing solution is also optimal under the new requirement. We can also use optimization history to tighten the cost bounds, or deduce that no solution can be found at a particular group.

5.5 Stacked Covering Subexpressions

Similar subexpressions can be shared at different levels. For example, a query may have two $CSEs$ $E_1 = A \bowtie B \bowtie C$ and $E_2 = B \bowtie C \bowtie D$. The two $CSEs$ share another smaller subexpression $E_3 = B \bowtie C$. It could be beneficial to compute E_3 first and use the result to compute E_1 and E_2. Their results are then used to compute other parts of the query. By extending our algorithm to CSE expression constructions, we automatically consider this kind of optimization strategy. We demonstrate the usage of stacked $CSEs$ in benchmark queries in Section 6.2.

6. EXPERIMENTAL RESULTS

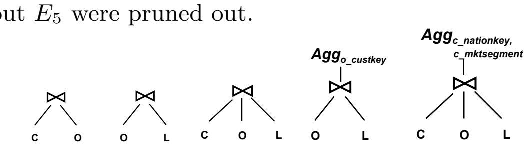
Our prototype implementation was built on Microsoft SQL Server. To demonstrate the benefits of exploiting similar subexpressions, we briefly outline several scenarios and describe experimental results. All experiments were performed on a workstation with a Pentium 4 3.0 GHz processor, 1GB of memory and one 160GB disk, running Windows XP. All queries were against a 1GB version (SF=1) of the TPC-H database.

When no candidate $CSEs$ are generated in *Step* 2, the only overhead is from collecting table signatures, and, if there are shared table signatures, attempting to generate candidates. We ran the optimizer on several TPC-H queries that have no sharing opportunities and tried to measure the overhead of our algorithm. The overhead was so small that we could not reliably measure it.

We cannot experimentally compare our approach to techniques proposed in previous work. It is simply not feasible for us to implement all of them in our system. But as detailed in Section 7, all of them consider only one or a small set of candidate $CSEs$ and none of them have a correct cost-based strategy to choose among multiple candidates.

6.1 A Query Batch

Our technique can detect and exploit similar subexpressions among the queries in a batch that are optimized and executed together. Our first experiment used a query batch consisting of the three queries in Example 1. Without pruning, the five candidate $CSEs$ shown in Figure 6 were generated in *Step 2.* Details about predicates and output columns are omitted and table names *customer*, *orders*, *lineitem* are abbreviated to C, O, L, respectively. With pruning enabled, all but E_5 were pruned out.



(a) E_1 (b) E_2 (c) E_3 (d) E_4 (e) E_5

Figure 6: Candidate CSEs for Example 1

Candidates E_1 to E_3 are, as expected, joins of different sets of tables. Candidate E_4 was generated because the optimizer considered preaggregation of the join of *orders* and *lineitem*. Candidate E_5 had a consumer in Q_3 too because the optimizer also considered preaggregation of the join result, followed by a join with *nation*, and final aggregation.

Applying the heuristics in Section 4.3 reduced the set of candidates to only E_5 (see below). E_1 was pruned out by Heuristic 1 because the join was too cheap. E_2, E_3, and E_4 were pruned out by Heuristic 4 because they were all contained by E_5 and E_5 produces the smallest result.

E_5:
```
select c_nationkey, c_mktsegment,
   sum(l_extendedprice) as vle, sum(l_quantity) as vlq
from customer, orders, lineitem
where c_custkey = o_custkey and o_orderkey = l_orderkey
and o_orderdate < '1996-07-01'
and c_nationkey > 0 and c_nationkey < 25
group by c_nationkey, c_mktsegment
```

With pruning disabled, all five candidates were given to the optimizer for consideration but with heuristic pruning enabled, only E_5 was considered. In both cases the optimizer chose the same final plan that used E_5 only. This verified that our heuristics pruned out the correct candidates and did not miss any optimization opportunities.

In the final plan E_5 is computed once and its result is used by all three queries as shown below (expressed in SQL).

Q'_1:
```
select * from E5
where c_nationkey > 0 and c_nationkey < 20
```

Q'_2:
```
select c_nationkey, sum(vle) as le, sum(vlq) as lq
from E5
where c_nationkey > 5 and c_nationkey < 25
group by c_nationkey
```

Q'_3:
```
select n_regionkey, sum(vle) as le, sum(vlq) as lq
from E5, nation
where c_nationkey = n_nationkey
and c_nationkey > 2 and c_nationkey < 24
group by n_regionkey
```

We compared three scenarios: regular optimization without $CSEs$, optimization using $CSEs$ with heuristic pruning, and optimization using $CSEs$ without heuristic pruning. For all three scenarios, we recorded the number of candidate $CSEs$ generated, number of additional CSE optimizations (in brackets), estimated cost of the chosen plan, and actual optimization and execution time, as shown in Table 1.

	No CSE	Using CSEs	Using CSEs (no heuristics)
# of CSEs [CSE Opts]	N/A	1 [1]	5 [15]
Optimization time (secs)	0.159	0.213	0.383
Estimated cost	539.93	206.47	
Execution time (secs)	165.54	55.64	

Table 1: Query batch (Q_1, Q_2, Q_3) in Example 1

With pruning enabled, clearly we only need one CSE optimization for one candidate. Without pruning enabled, all five candidate $CSEs$ are competing against each other. Nevertheless, our optimization algorithm in Section 5.3 reduces the number of optimizations down to 15 (from 31). Most of these optimizations are cheap because thay exploit previous optimization history.

We achieve close to a 3X reduction in execution time with a modest increase in optimization time. Applying heuristic pruning significantly reduced the optimization overhead. The overall increase in optimization time is negligible compared with the savings in execution time.

In this example there are many different ways for a user to rewrite the queries using `WITH` clauses. In fact, each candidate in Figure 6 can be written using a `WITH` clause. But only one rewrite (using E_5) achieves optimal performance. This illustrates the danger of relying on user-defined `WITH` clauses to find the best common subexpressions. An optimizer can consider all options and choose among them in a cost-based manner.

6.2 Stacked CSEs

The optimal choice of $CSEs$ can be quite different with a slightly different query batch. In the second experiment, we added another query Q_8 to the query batch in the previous experiment.

Q_8:
```
select p_type, sum(p_availqty) as qty
from part, orders, lineitem
where p_partkey = l_partkey and o_orderkey = l_orderkey
and o_orderdate < '1996-07-01'
order by p_type
```

Without pruning, the same set of five candidate $CSEs$ shown in Figure 6 were generated. Enabling pruning reduced the set of candidates to E_2 and E_5. E_2 could not be pruned out because Q_8 contains a potential consumer for it so that E_2 was no longer fully contained by E_5.

The final plan used both E_2 and E_5. The result of E_2 is used to answer Q_8 and, more interestingly, it is also used to compute E_5. The result of E_5 is then used to compute the first three queries in the same way as before. We show E_2, the new V'_5 and Q'_8 in SQL below.

E_2:
```
select o_custkey, l_partkey, l_extendedprice, l_quantity
from orders, lineitem
where o_orderkey=l_orderkey and o_orderdate<'1996-07-01'
```

E'_5:
```
select c_nationkey, c_mktsegment,
   sum(l_extendedprice) as vle, sum(l_quantity) as vlq
from customer, E2
where c_custkey = o_custkey
and c_nationkey > 0 and c_nationkey < 25
group by c_nationkey, c_mktsegment
```

Q'_8:
```
select p_type, sum(p_availqty) as qty
from part, E2
where p_partkey = l_partkey
group by p_type
```

	No CSE	Using CSEs	Using CSEs (no heuristics)
# of CSEs [CSE Opts]	N/A	2 [1]	5 [7]
Optimization time (secs)	0.215	0.321	0.518
Estimated cost	716.03	372.06	
Execution time (secs)	216.40	85.94	

Table 2: Query batch (Q_1, Q_2, Q_3, Q_8)

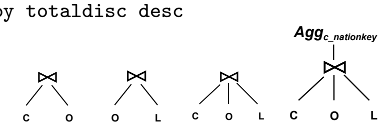
Table 2 shows the results for the query batch of Q_1, Q_2, Q_3, and Q_8. Exploiting similar subexpressions greatly reduces the execution time. The additional query results in a different overall choice of covering subexpressions, which confirms the importance of full cost-based optimization.

6.3 Nested Subquery

A complex decision-support query may contain several subqueries that are similar, providing great opportunities for exploiting similar subexpressions between subqueries and the main query block or among different subqueries.

We ran an experiment with a nested query that is similar to Query 11 in the TPC-H benchmark [2]. Both the main query and the subquery contain a join of *customer*, *orders*, and *lineitem*, although they provide different aggregated values and output different columns.

Q_9:
```
select c_nationkey, n_name, sum(l_discount)
from customer, orders, lineitem, nation
where c_custkey = o_custkey and o_orderkey = l_orderkey
and c_nationkey = n_nationkey
group by c_nationkey, n_name
having sum(l_discount) > (
 select sum(l_discount) / 25
 from customer, orders, lineitem
 where c_custkey = o_custkey and o_orderkey = l_orderkey)
order by totaldisc desc
```



(a) E_1 (b) E_2 (c) E_3 (e) E_4

Figure 7: Candidates for Q_9

Without heuristic pruning the system generated four candidate $CSEs$, as shown in Figure 7, but only E_4 was used in the final plan. With heuristic pruning enabled, only E_4 was generated and also used in the same final plan. E_1 was pruned out by Heuristic 1 while E_2 and E_3 were pruned out by Heuristic 4. Again, our heuristics pruned out the correct candidates. We show E_4 and the rewritten query below.

E_4:
```
select c_nationkey, sum(l_discount) as totaldisc
from customer, orders, lineitem
where c_custkey = o_custkey and o_orderkey = l_orderkey
group by c_nationkey
```

Q'_9:
```
select c_nationkey, n_name, totaldisc
from E4, nation
where c_nationkey = n_nationkey
having totaldisc > (select sum(totaldisc)/25 from E4)
order by totaldisc desc
```

Table 3 shows the results with and without using $CSEs$ (with pruning enabled). In this case, we cut execution time by half, again with a modest increase in optimization time.

	No CSE	Using CSEs	Using CSEs (no heuristics)
# of CSE [CSE Opts]	N/A	1 [1]	4 [8]
Optimization time (secs)	0.138	0.197	0.295
Estimated cost	442.61	240.49	
Execution time (secs)	135.26	67.67	

Table 3: Nested Query

6.4 Materialized View Maintenance

A database may contain many similar materialized views. Every affected materialized view has to be maintained after an update, so it may be possible to reduce maintenance overhead by optimizing the maintenance expressions together and exploiting similar subexpression.

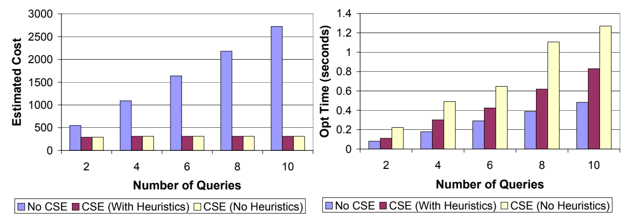
Our technique can be applied to optimizing view maintenance plans. When a base table is updated, the updated tuples are stored in an internal work table, called a *delta* table, and the table is then used to drive maintenance for all affected views. We treat the delta table as a special table when generating table signatures and constructing $CSEs$.

To verify our prototype, we created three materialized views whose expression are the same as the three queries in Example 1. When updating the *customer* table, maintenance time was reduced by a factor of three using a CSE similar to E_5 in Example 1. We omit the details due to space limitation.

6.5 Scaleup Analysis

We also conducted experiments to assess the performance of our approach for increasing number of queries and for larger queries.

First, we consider the effect of increasing the number of queries. We created several query batches with different number of queries. Similar to Q_1, Q_2 or Q_3, each query contains joins of tables *lineitem*, *orders*, and *customer*, but has different local predicates, group on different columns, and may also join additional tables *nation* and *region*.



(a) Estimated Cost (b) Optimization Time

Figure 8: Optimization of Query Batches

Figure 8 shows estimated costs and optimization time for query batches with two to ten queries. With pruning enabled, one or two candidate $CSEs$ were generated, compared to four or five candidates without pruning. In either case, a single CSE was used in the final plan for each query batch. As expected, the cost benefit is proportional to the number of queries in the batch. With pruning enabled, the CSE optimization overhead is very small and the optimization time increases linearly with the number of queries in the batch. The results indicate that our approach scales well with increasing batch sizes.

	No CSE	Using CSEs	Using CSEs (no heuristics)
# of CSEs [CSE Opts]	N/A	2 [2]	51 [391]
Optimization time (secs)	2.103	3.892	12.745
Estimated Cost	294.57	173.45	
Execution time (secs)	81.45	48.73	

Table 4: Complex Joins

Second, we verified how well our approach scales up with query size. We used a query batch consisting of two queries. Each query joins all eight tables in TPC-H and finally aggregates by *region*. Each query also contains different local predicates. Without heuristics enabled, 51 candidate $CSEs$ were generated. Most of the candidates were either cheap or contained by other candidates. By exploiting the fact that some candidates are independent of each other, the number of optimizations can be significantly reduced. With heuristics enabled, only two candidates were generated and two optimizations were performed. Using $CSEs$, we achieved almost a 2X reduction in plan costs with a modest increase in optimization time.

[2]Query 2 and 15 are also similar nested queries but they can be computed cheaply and thus are of less interest.

7. RELATED WORK

There is a large body of research on multi-query optimization [3, 10, 13, 15, 14, 16]. The idea of exploiting similar subexpressions has also been applied to materialized view selection [8, 9, 12] and optimization of queries with nested subqueries [11]. In this paper, we propose a uniform solution to all three problems.

Early work on multi-query optimization [10, 15, 14] focused primarily on expensive exhaustive algorithms and the solutions were not integrated with the system's query optimizer. The work in [3, 16] was limited to finding common subexpressions in a post-optimization phase and considering sharing opportunities only among the best plans for each query. This can obviously lead to suboptimal plans.

Roy et al. [13] were the first to describe integration of multi-query optimization features into a Volcano-style optimizer. However, the proposed solution is somewhat limited and may miss certain important optimization opportunities. First, every covering expression is constructed to cover *all* its potential consumers, which, as observed in Section 4.3.2, may result in expressions producing large results. Second, their optimization algorithm does not consider multiple competing covering expressions correctly and may incorrectly prune out an alternative based on usage cost alone. The proposed greedy algorithm does not always produce an optimal solution. For example, it misses the case when using either covering expressions e_2 or e_3 alone is less efficient than using e_1, but using *both* e_2 and e_3 is more beneficial than using e_1. Finally, it requires extensive and fundamental modifications to the optimizer, something database vendors are reluctant to do because of cost and quality concerns.

Maintenance cost for a set of materialized views can sometimes be reduced by creating supporting materialized views. Ross et al. [12] considered how to determine the best set of supporting views given a limited amount of space. Mistra et al. [9] applied multi-query optimization techniques to speed up maintenance of a set of views. Lehner et al. [8] also considered exploiting similar subexpressions when maintaining multiple materialized views. They create the widest possible covering subexpression and force each consumer to use it in the final plan. The covering subexpression is also designed to cover all potential consumers. The paper does not discuss how to consider multiple competing covering expressions. Folkert et al. [4] proposed a refresh scheduling algorithm such that materialized views can be refreshed using query rewrite against previously refreshed materialized views. Our solution can achieve the same improvement but is much more general. The common subexpressions can be either the views themselves or part of them.

Rao and Ross [11] studied the problem of exploiting invariant parts of a nested subquery. Our technique can be applied to nested queries and achieve the same effect.

Automated selection of indexes and views for a given workload is described in [1]. Our solution automatically considers using existing indexes and views in order to generate the optimal plan. Queries can benefit from exploiting similar subexpressions, no matter whether the workload information is available.

8. CONCLUSION

In this paper, we present a practical, scalable, and uniform solution to detecting and exploiting similar subexpressions to improve query performance. It is applicable to all similar subexpressions no matter whether they originate from a single query, multiple queries or view maintenance expressions. Our table signature technique finds potentially sharable subexpressions very efficiently. There is virtually no overhead when queries do not have any sharable expression. We consider all possible covering subexpressions, including popular (with most consumers) ones and less popular (with fewer consumers) ones, wide (more tables) ones and narrow (fewer tables) ones, etc. The query optimizer evaluates different candidates in fully cost-based manner.

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *Proc. of VLDB Conference*, 2000.

[2] D. DeHaan, P.-Å. Larson, and J. Zhou. Stacked indexed views in microsoft sql server. In *Proceedings of SIGMOD Conference*, 2005.

[3] S. J. Finkelstein. Common subexpression analysis in database applications. In *Proceedings of SIGMOD Conference*, 1982.

[4] N. Folkert, A. Gupta, A. Witkowski, S. Subramanian, S. Bellamkonda, S. Shankar, T. Bozkaya, and L. Sheng. Optimizing refresh of a set of materialized views. In *Proceedings of VLDB Conference*, 2005.

[5] J. Goldstein and P.-Å. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *Proceedings of SIGMOD Conference*, 2001.

[6] G. Graefe. The Cascades framework for query optimization. *Data Engineering Bulletin*, 18(3), 1995.

[7] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceeding of ICDE Conference*, 1993.

[8] W. Lehner, R. Cochrane, H. Pirahesh, and M. Zaharioudakis. fAST refresh using mass query optimization. In *Proc. of ICDE Conference*, 2001.

[9] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *Proceedings of SIGMOD Conference*, 2001.

[10] J. Park and A. Segev. Using common subexpressions to optimize multiple queries. In *Proceedings of ICDE Conference*, 1988.

[11] J. Rao and K. A. Ross. Reusing invariants: A new strategy for correlated queries. In *Proceedings of SIGMOD Conference*, 1998.

[12] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proceedings of SIGMOD Conference*, 1996.

[13] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proc. of SIGMOD Conference*, 2000.

[14] T. Sellis and S. Ghosh. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, 2(2), 1990.

[15] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1), 1988.

[16] S. N. Subramanian and S. Venkataraman. Cost-based optimization of decision support queries using transient views. In *Proceedings of SIGMOD Conference*, 1998.