Optimization of Common Table Expressions in MPP Database Systems

Amr El-Helw*, Venkatesh Raghavan*, Mohamed A. Soliman*, George Caragea*,
Zhongxian Gu†, Michalis Petropoulos‡

* Pivotal Inc. Palo Alto, CA, USA
† Datometry Inc. San Francisco, CA, USA
‡ Amazon Web Services Palo Alto, CA, USA

ABSTRACT

Big Data analytics often include complex queries with similar or identical expressions, usually referred to as Common Table Expressions (CTEs). CTEs may be explicitly defined by users to simplify query formulations, or implicitly included in queries generated by business intelligence tools, financial applications and decision support systems. In Massively Parallel Processing (MPP) database systems, CTEs pose new challenges due to the distributed nature of query processing, the overwhelming volume of underlying data and the scalability criteria that systems are required to meet. In these settings, the effective optimization and efficient execution of CTEs are crucial for the timely processing of analytical queries over Big Data. In this paper, we present a comprehensive framework for the representation, optimization and execution of CTEs in the context of *Orca* – Pivotal's query optimizer for Big Data. We demonstrate experimentally the benefits of our techniques using industry standard decision support benchmark.

1. INTRODUCTION

Big Data analytics are becoming increasingly common in many business domains, including financial corporations, government agencies, and insurance providers. The uses of Big Data range from generating simple reports to executing complex analytical workloads. The increase in the amount of data being stored and processed in these domains exposes many challenges with respect to scalable processing of analytical queries. Massively Parallel Processing (MPP) databases address these challenges by distributing storage and query processing across multiple nodes and processes.

Common Table Expressions (CTEs) are commonly used in complex analytical queries that often have many repeated computations. A CTE can be seen as a temporary table that exists just for one query. The purpose of CTEs is to avoid re-execution of expressions referenced more than once within a query. CTEs may be defined explicitly, or generated implicitly by the query optimizer (cf. Section 8). The following example illustrates a use case of CTEs defined explicitly using the SQL `WITH` clause:

Example 1. Consider the following query:

```
WITH v as (SELECT i_brand, i_current_price, max(i_units) m
           FROM item
           WHERE i_color = 'red'
           GROUP BY i_brand, i_current_price)
SELECT * FROM v WHERE m < 100
AND v.i_current_price IN (SELECT min(i_current_price)
                          FROM v WHERE m > 5);
```

Example 1 includes a CTE (v), with filtering and grouping operations, where v is referenced twice in the main query. This is an alternative to repeating the whole expression that defines v. [1]

In practice, the definition of v may be much more complex, containing joins, subqueries, user-defined functions, etc. In addition, it may be referenced more than twice in the query. Thus, defining it as a CTE achieves two goals: (i) simplifying the query, making it more readable, and (ii) if handled carefully, performance gains could be achieved by evaluating a complex expression only once.

CTEs follow a producer/consumer model, where the data is produced by the CTE definition, and consumed in all the locations where that CTE is referenced. One possible approach to execute CTEs is to expand (inline) all CTE consumers, essentially rewriting the query internally to replace each reference to the CTE with the complete expression. This approach simplifies query execution logic, but may incur performance overhead due to executing the same expression multiple times. In addition, query optimization time could increase if the expanded query is complex.

An alternative approach is to execute CTEs in a true producer/consumer fashion, where the CTE expression is separately optimized and executed only once, the results are kept in memory, or written to disk if the data does not fit in memory or has to be communicated between different processes – as is the case in MPP systems. The data is then read whenever the CTE is referenced. This approach avoids the cost of repeated execution of the same expression, although it may incur an overhead of disk I/O. The impact of this approach on query optimization time is rather limited, since the optimizer chooses one plan to be shared by all CTE consumers. However, important optimization opportunities could be missed due to fixing one execution plan for all consumers.

1.1 Challenges

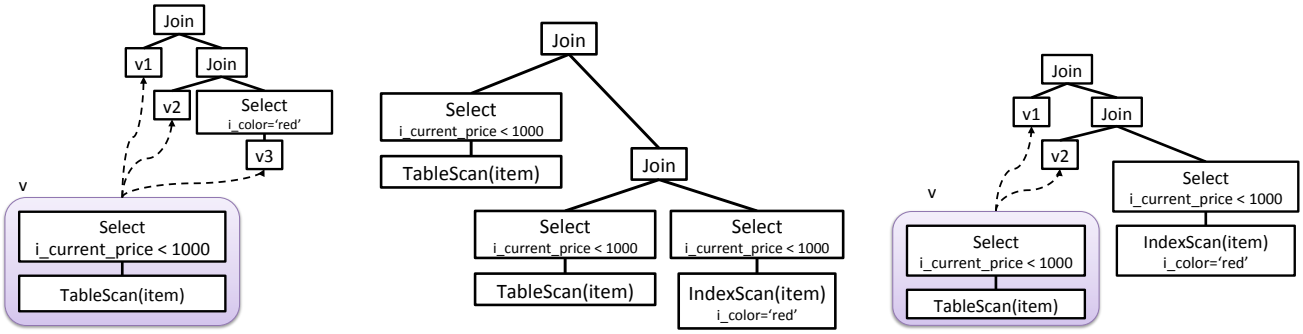
In this section, we highlight the main challenges we tackle.

1.1.1 Deadlock Hazard

MPP systems leverage parallel query execution, where different parts of the query plan execute simultaneously as separate processes, possibly running on different machines. Data flows between these processes as the operators in the plan are executed. In some

[1] v could be used in multiple queries if defined using `CREATE VIEW` statement. In this case, the query parser can automatically include v definition in the parse tree of any query that references v.

Proceedings of the VLDB Endowment, Vol. 8, No. 12



(a) No inlining (b) Inline all CTEs (c) Partial inlining

Figure 1: Possible plans for the query in Example 2

cases, a process has to wait until another process produces the data it needs. For example, if the CTE definition is in one process and the CTE consumer is in a different process, then the latter has to wait for the former. For complicated queries involving multiple CTEs, the optimizer needs to guarantee that no two or more processes could be waiting on each other during query execution. CTE constructs need to be cleanly abstracted within the query optimization framework to guarantee deadlock-free plans. We discuss how our design greatly simplifies deadlock handling in Sections 4 and 5.

1.1.2 Enumerating Inlining Alternatives

The approaches of always inlining CTEs, or never inlining CTEs, can be easily proven to be sub-optimal, as we show here.

Example 2. Given the `item` table from TPC-DS [13], and assuming that we have an index on `item.i_color`, consider the following query:

```
WITH v as (SELECT i_brand, i_color FROM item
           WHERE i_current_price < 1000)
SELECT v1.*
FROM v v1, v v2, v v3
WHERE v1.i_brand = v2.i_brand AND v2.i_brand = v3.i_brand
AND v3.i_color = 'red';
```

Figure 1(a) illustrates the plan produced when the CTE is never inlined. In this scenario, the CTE is executed once, and its results are reused 3 times. This approach avoids repeated computation. However, this approach does not take advantage of the index on `i_color`. The opposite approach is illustrated in Figure 1(b), where all occurrences of the CTE are replaced by the expansion of the CTE expression. This allows the optimizer to utilize the index on `i_color` in one of the inlined expressions. However, it suffers from the repeated computation of the other two inlined expressions. Figure 1(c) depicts a possible plan in which one occurrence of the CTE is expanded, allowing the use of the index, while the other two occurrences are not inlined, to avoid recomputing the common expression. This plan would not be considered by adopting the inlining/no-inlining approaches in isolation. The query optimizer needs to efficiently enumerate and cost plan alternatives that combine the benefits of these approaches. We show how our plan enumeration approach addresses this challenge in Section 6.

1.1.3 Contextualized Optimization

CTEs should not be optimized in isolation without taking into account the context in which they occur. Isolated optimization can easily miss several optimization opportunities. For example, if the query has filters on all CTE consumers, these filters can possibly be pushed inside the CTE plan to reduce the number of tuples to be produced. Similarly, multiple consumers may require CTE results to be partitioned or sorted in the same way. Considering plan alternatives that push such requirements into the CTE plan can lead to avoiding re-sorting/re-partitioning the same data. However, re-optimizing a CTE every time it is referenced in the query does not scale, and causes the search space to grow exponentially. CTE optimization needs to be handled organically by the optimizer to consider their interplay with other optimizations and allow early-pruning of inferior plan alternatives. We show how we tackle this challenge in Section 7.

1.2 Contributions

We present a novel approach for representing, optimizing and executing queries with non-recursive CTEs. Our approach is implemented in Orca, the Pivotal Query Optimizer [12], and is currently used in production. Our contributions are summarized as follows:

- A novel framework for the optimization of CTEs in MPP database systems. Our framework extends and builds upon our optimizer infrastructure to allow optimization of CTEs within the context where they are used in a query
- A new technique in which a CTE does not get re-optimized for every reference in the query, but only when there are optimization opportunities, e.g. pushing down filters or sort operations. This ensures that the optimization time does not grow exponentially with the number of CTE consumers
- A cost-based approach for deciding whether or not to expand CTEs in a given query. The cost model takes into account disk I/O as well the cost of repeated CTE execution
- Several optimizations that reduce the plan search space and speed up query execution, including pushing down predicates into CTEs, always inlining CTEs if referenced only once, and eliminating CTEs that are never referenced
- A query execution model that guarantees that the CTE producer is always executed before the CTE consumer(s). In MPP settings, this is crucial for deadlock-free execution

We have also conducted an experimental evaluation using TPC-DS benchmark [13] to demonstrate the efficiency of our techniques. The rest of the paper is organized as follows: Section 2 reviews related work, and Section 3 provides the necessary background on MPP architecture and query optimization in Orca. Section 4 outlines our proposed CTE representation, and Section 5 demonstrates how this representation guarantees deadlock-free query execution.

Section 6 describes our plan enumeration technique. Section 7 discusses property derivation and enforcement as well as costing for the plan alternatives. Section 8 discusses how CTEs can be used as an optimization tool for queries with no explicit CTEs. Section 9 describes how queries with CTEs are executed in an MPP system. Section 10 presents our experimental evaluation which demonstrates the performance gain of our approach. Lastly, Section 11 concludes the paper.

2. RELATED WORK

The problem of optimizing common subexpressions has been well studied in the domain of query processing and optimization. We focus on two important areas that are pertinent to our proposal.

CTE Optimization. Silva et al. [11] proposed an extension to the SCOPE query optimizer [15], where a 2-phase optimization technique is used to address the contextualized optimization challenge discussed in Section 1.1.3. The first phase uses the original SCOPE optimizer with an additional step that records the required physical properties (e.g. data partitioning and sorting) of all CTE consumers in linked lists. Then, a subsequent re-optimization phase is used to identify the least common ancestors of CTE consumers, and re-optimize CTE producers based on the CTE consumers' local requirements, with the goal of identifying a globally-optimal plan.

In contrast to [11], our proposal integrates CTE optimization organically at the core of the query optimizer, eliminating the need for re-optimization (cf. Section 7). Our representation framework also allows identifying the optimization entry point of CTEs, without the need to search for least common ancestors. Finally, by conducting optimization in one phase, our method lends itself naturally to pruning the plan space early on to avoid unnecessary optimization.

PostgreSQL [1] views CTEs as a means to isolate a subquery within a complex query. The generated plan evaluates a CTE in a separate subplan, optimized in isolation of the main query. This approach could result in dismissing important optimization opportunities such as (1) inlining CTE, (2) enforcing physical properties such as sort order on CTE output when all the CTE consumers require the same properties, and (3) pushing down predicates into the CTE subplan (cf. Section 6.3).

Oracle optimizer [4] generates plans that can store the results of a subquery into a temporary table that can be referred to as many times as needed. The optimizer can inline each reference to the refactored subquery. The MATERIALIZE and INLINE optimizer hints can be used to influence the decision. HP Vertica [3] and PDW [10] also maintain CTE results in a temporary table for the duration of query execution.

Optimization using Materialized Views. In traditional database systems, using materialized views in query optimization and execution is a well-studied problem [6, 14, 8]. Our approach of exploring inlining alternatives has similarities to the cost-driven methodology used in deciding whether or not to use a materialized view in answering a query. However, the problem addressed in our work is orthogonal to the materialized view selection problem and can be differentiated as follows; first, using materialized views depends primarily on view matching techniques to decide whether a materialized view can be used to optimize a particular query. Explicit CTEs do not generally require view matching since they are defined and referenced in the query. View matching techniques can be used to detect common subexpressions that are implicitly defined in the query. Our framework can leverage and build on such techniques to capture and optimize subexpressions within our CTE framework (cf. Section 8.2). Second, as we show in Section 7, CTEs can take advantage of contextual optimization, where CTE consumers' local requirements may impose new plans onto the CTE producer side, e.g., pushing predicates and/or sort orders from the consumer to the producer. This is not applicable to materialized views because the creation of the materialized view happens separately from the query that may utilize it. Furthermore, unlike materialized views, CTEs are defined as part of the query and are not stored. Hence the problems of materialized view maintenance and design do not apply.

3. BACKGROUND

In this section, we outline two topics that are key to this work. In Section 3.1, we describe the underlying MPP architecture, while in Section 3.2, we give an overview on Orca query optimizer.

3.1 Massively Parallel Processing

Modern scale-out database engines are usually based on one of two design principles: *sharded* and *massively parallel processing (MPP)* databases. Both are shared-nothing architectures, where each node manages its own storage and memory, and they are typically based on horizontal partitioning. Common use cases for sharding include "west" vs "east" customers, or partitioning user names based on ranges of the alphabet. Sharded systems optimize for executing queries on small subsets of the shards, and communication between shards is relatively limited. Shards can be placed in different data centers, or even geographies.

MPP databases optimize for parallel execution of each query. The nodes are usually collocated within the same data center, and each query can access data across all the nodes. A query optimizer generates an execution plan that includes explicit data movement directives, and the cost of moving data is taken into account during optimization. A query executing in an MPP database can include several pipelined execution stages, with explicit communication between nodes at each stage. For example, a multi-stage aggregation can be used to compute an aggregate over the entire dataset using all the nodes.

Pivotal's Greenplum Database (GPDB) [9] is an MPP analytics database. GPDB adopts a shared-nothing architecture with multiple cooperating processors (typical deployments include tens or hundreds of nodes). Storage and processing of large amounts of data are handled by distributing the load across several servers or hosts to create an array of individual databases, working together to present a single database image. The *master* is the entry point, where clients connect and submit SQL statements. The master coordinates work with other database instances, called *segments*.

During query execution, data can be distributed in multiple ways including *hashed* distribution, where tuples are distributed to segments based on some hash function, *replicated* distribution, where a full copy of a table is stored at each segment and *singleton* distribution, where the whole distributed table is gathered from multiple segments to a single host (usually the master).

Special operators, called *Motion* operators, are used to accomplish data communication among segments. A Motion operator acts as the boundary between two active processes sending/receiving data and potentially running in different nodes. The goal of Motion operators is to establish a given data distribution. For example, to establish a hashed distribution on column `x`, an instance of Redistribute(`x`) Motion operator, running on segment S, sends tuples on S to other segments based on the hash value of `x`, and also receives tuples from other Redistribute(`x`) operator instances running, in parallel, on other segments. Similarly,

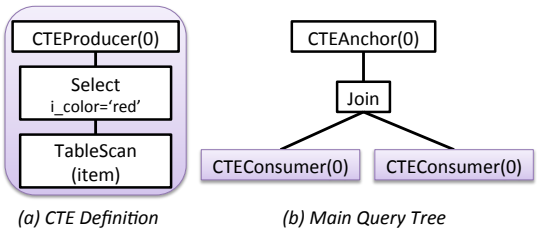


Figure 2: Logical representation of query in Example 3

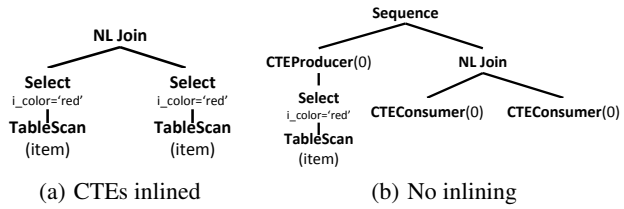


Figure 3: Execution plans of query in Example 3

a Broadcast Motion and Gather Motion operators are used to establish *replicated* and *singleton* distributions, respectively. In Section 7.1, we elaborate on how Motion operators are used to enforce required data distribution during query optimization.

3.2 Query Optimization in Orca

Orca [12] is the query optimizer of Pivotal data management products, including GPDB [9] and HAWQ [5]. Orca is a modern top-down query optimizer based on the Cascades framework [7].

In Orca, the plan space is encoded in a compact data structure called the *Memo* [7] consisting of a set of containers called *groups*. Each group contains logically equivalent expressions, called *group expressions*. Each group expression is an operator whose children are other groups. This recursive structure of the Memo allows compact encoding of a huge space of possible plans. The Memo group that contains the query top operator is called the "root" group.

Plan alternatives are generated by transformation rules that produce either equivalent logical expressions, or physical implementations of existing expressions. The results of applying transformation rules are added to the Memo, which may result in creating new groups and/or adding new expressions to existing groups.

During optimization, an operator may request its children to satisfy physical properties (e.g., sort order and data distribution). A child plan may either satisfy the required properties on its own (e.g., an IndexScan delivers sorted data), or an *enforcer* operator (e.g., Sort) needs to be used to deliver the required properties.

4. REPRESENTATION OF CTEs

To illustrate how we represent queries with CTEs in Orca, we start with the following simple example:

Example 3.

```
WITH v AS (SELECT i_brand FROM item WHERE i_color = 'red')
SELECT * FROM v as v1, v as v2
WHERE v1.i_brand = v2.i_brand;
```

The initial logical representation of the query is shown in Figure 2. We introduce the following new CTE operators:

- ***CTEProducer***: This operator is initially set as the root of a separate logical tree which corresponds to the CTE definition. There is one such tree – and one such CTEProducer operator – for every CTE defined in the query. These trees are not initially connected to the main logical query tree. Each CTEProducer has a unique id.
- ***CTEConsumer***: This operator denotes the place in the query where a CTE is referenced. The number of CTEConsumer nodes is the same as the number of references to CTEs in the query. The id in the CTEConsumer operator corresponds to the CTEProducer to which it refers. There can be multiple CTEConsumers referring to the same CTEProducer.

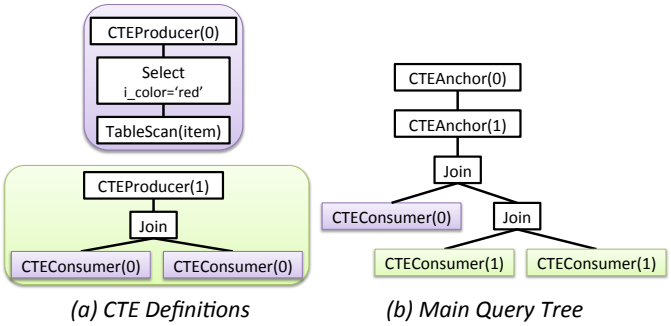


Figure 4: Logical representation of query in Example 4

- ***CTEAnchor***: This operator denotes where a particular CTE is defined in the query. It defines the scope of that CTE. A CTE can be referenced only in the subtree rooted by the corresponding CTEAnchor operator.
- ***Sequence***: This is a binary operator that executes its children in order (left to right), and returns the results of the right child. Orca also uses the Sequence operator for optimizing queries on partitioned tables [2].

Figure 3 shows two possible execution plans for the query in Figure 2. In the first plan, all CTEs are inlined. In this case, the CTEAnchor is removed, and each CTEConsumer is replaced with the whole tree representing the CTE definition. In the second plan, there is no CTE inlining. The CTEAnchor has been replaced by a Sequence operator that has the CTEProducer as its first child, and the original child of the CTEAnchor as its second child.

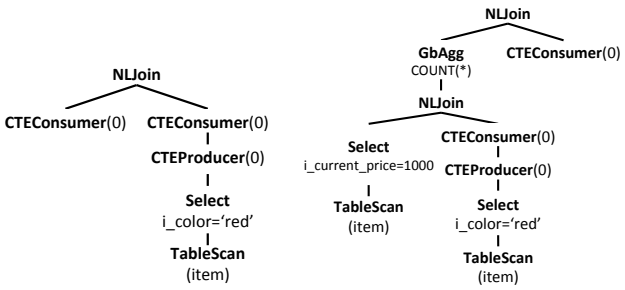
The Sequence operator guarantees a specific order of execution, where the subtree under the CTEProducer is executed first before any of the corresponding CTEConsumers start execution. As a result, when execution reaches the CTEConsumer, the data is already available to read. This guarantees that generated plans have no deadlocks, especially with plans that have multiple CTEs. We elaborate on this point in Section 5.

The previous operators can also be used to represent nested CTEs, as we show in the following example:

Example 4.

```
WITH v as (SELECT i_current_price p FROM item
           WHERE i_color = 'red'),
     w as (SELECT v1.p FROM v as v1, v as v2
           WHERE v1.p < v2.p)
SELECT * FROM v as v3, w as w1, w as w2
WHERE v3.p < w1.p + w2.p;
```

Figure 4 shows the logical representation of the query in Example 4. Each CTEProducer node sits on top of the logical tree of the corresponding CTE. The main query has two CTEAnchor nodes, in the same order as the occurrence of the CTEs in the WITH clause.



(a) Possible execution plan for query in Example 3

(b) Execution plan dependent on input data

Figure 5: Execution plans with deadlocks

Note that some CTEConsumer nodes are in the main query and some are inside the tree under a CTEProducer, which corresponds to CTEs being referenced from inside other CTEs.

5. CTE EXECUTION AND DEADLOCKS

Executing plans with CTEs creates dependencies between different parts of the plan that need to be satisfied at runtime. The optimizer must take these dependencies into account, and guarantee that the generated plan is deadlock free for any possible input.

A possible approach is to place the CTEProducer as a child of one of the CTEConsumer nodes. Consider the query in Example 3, with the logical representation in Figure 2. One possible execution plan can use a Nested Loop Join (NLJoin), and attach the CTEProducer directly under the CTEConsumer on the inner (right) side of the join (see Figure 5(a)). In this plan, the NLJoin node triggers the execution of its outer (left) child, which triggers the execution of the CTEConsumer on the left side. This CTEConsumer would be blocked, since the tuples it tries to read have not yet been produced. Since the outer child of the NLJoin is blocked, execution never reaches the inner child, which is supposed to execute the CTEProducer. Therefore this plan results in a deadlock. This deadlock could have been avoided if the CTEProducer is placed under the consumer which is executed first, which is the outer (left) child of the NLJoin in this case. A naïve method to avoid deadlocks can be summarized in the following steps:

1. Optimize query without considering the CTE expressions.
2. Optimize each of these CTE expressions separately.
3. For each CTE in the query (in order of dependency):
   (a) Traverse the execution plan of the main query in the order of execution
   (b) Plug the tree corresponding to this CTEProducer under the first corresponding consumer encountered during this traversal

One obvious drawback of this approach is that the first step does not take into account the cost of executing the CTEProducer, which means that the plan chosen for the main query may not be optimal anymore after plugging in the smaller CTE subplans.

An additional source of complexity with this approach comes from the fact that executing some parts of a query plan might be skipped altogether for certain inputs. Consider the plan in Figure 5(b). In this plan, the CTEProducer was placed according to the algorithm above, under the first CTEConsumer encountered in the order of execution. Assume now that the filter on `i_current_price` does not return any tuples. Most engines will then optimize execution and simply skip executing the inner side of the join, since the join operator will not produce any tuples. In this case, the CTEProducer placed under the inner child of the join will never be executed. When execution reaches the CTEConsumer placed under the top join operator, it will try to read tuples from a CTEProducer that has never executed, resulting in a deadlock.

In theory, these situations could be avoided during query optimization or execution. However, this would make the processing logic increasingly complex by tightly coupling optimizer and execution engine designs, resulting in maintainability and extensibility challenges. The problem is aggravated in the context of MPP systems, where we need to handle communication among processes on different nodes. The Sequence operator greatly simplifies the decision on the placement of the CTEProducer in the plan, and allows the optimizer to transparently perform several optimizations without worrying about deadlocks.

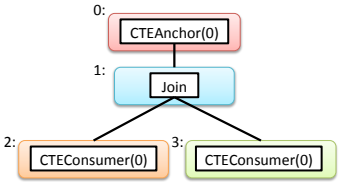


Figure 6: Memo after inserting the query in Figure 2

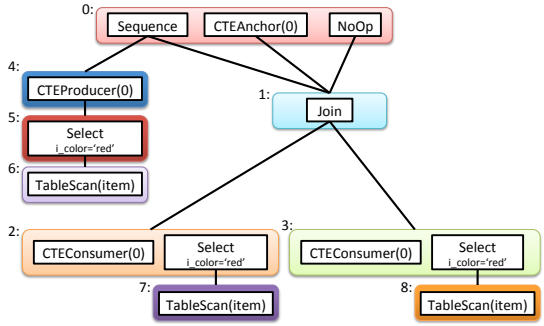


Figure 7: Memo after applying transformations

6. PLAN ENUMERATION

In this section, we illustrate how to generate plan alternatives for queries with CTEs. We use the logical query depicted in Figure 2 for illustration. The initial logical query expression is first inserted into the Memo, creating as many Memo groups as required. Figure 6 shows a representation of the Memo after initializing it with this logical expression, where each numbered box represents a distinct Memo group.

6.1 Transformation Rules

A transformation rule takes as input an expression in a Memo group, and produces another expression to be added to the same group. For each CTE, we generate the alternatives of inlining or not inlining the CTE. Figure 7 depicts the Memo after applying the following CTE-related transformations:

- The first rule is applied to the CTEAnchor operator. It generates a Sequence operator in the same group as the CTEAnchor (group 0), such that the left child of the Sequence is the whole tree representing the CTE definition – creating as many new groups as necessary (groups 4, 5, and 6) – and the right child of the Sequence is the original child of the CTEAnchor (group 1).

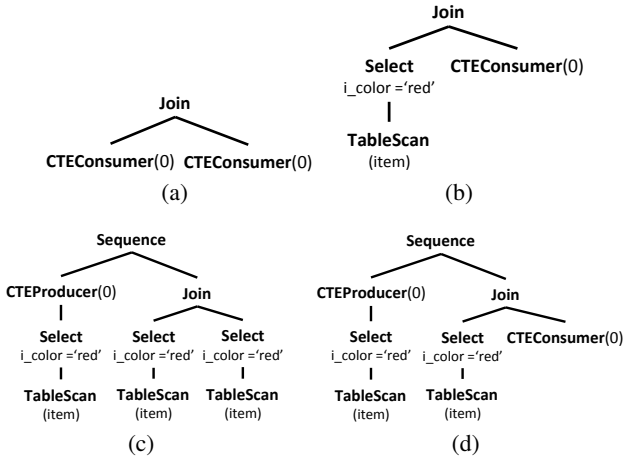


Figure 8: Other possible plans

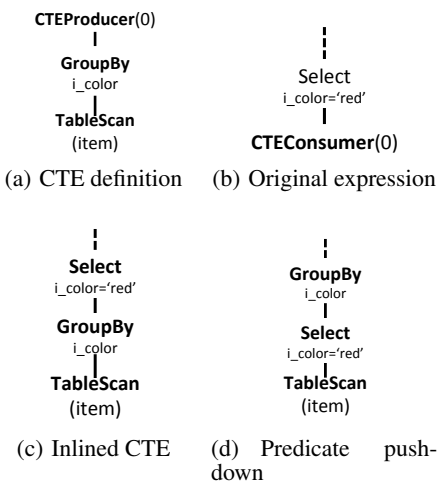


Figure 9: Pushing down predicates through inlined CTE

- The second rule is also applied to the CTEAnchor, generating a NoOp operator in the same group (group 0), with its only child being the child of the CTEAnchor (group 1).
- The third rule is applied to CTEConsumer operators, generating a copy of the CTE definition, and adding that expression to the same group as the CTEConsumer. For example, for the CTEConsumer in group 2, the CTE definition is added so that the Select operator is also in group 2, and its child (the TableScan operator) is added to a new group (group 7).

After estimating the cost of the different alternatives (cf. Section 7), the optimizer chooses the alternative with the lowest cost from each group. For example, if the optimizer chooses a plan rooted by the NoOp operator from group 0 with the inlined expressions from groups 2 and 3, we get the plan in Figure 3(a). Alternatively, if the optimizer picks a plan rooted by the Sequence operator from group 0 with the CTEConsumers from groups 2 and 3, we end up with the plan in Figure 3(b).

Figure 8 shows other possible plans that can be picked by choosing other operators. Note, however, that not all the plans shown in this Figure are valid. For example, the plans in Figures 8(a) and 8(b) contain CTEConsumer operators without a corresponding CTEProducer. These plans cannot be executed since the CTEConsumers need to read data that is never produced. The plan in Figure 8(c) has a CTEProducer without any corresponding CTEConsumers. This means that the CTE expression would be needlessly executed one additional time and cached. We avoid generating such plans using the algorithm explained in Section 6.2.

The plan in Figure 8(d) is not an invalid plan. However, it is not the most efficient plan, since it contains only one CTEConsumer that corresponds to the included CTEProducer. This plan is almost equivalent to the plan in Figure 3(a) in terms of cost, except that it incurs the additional overhead of caching the CTE outputs. We avoid generating such plans as we explain later in Section 6.3.2.

Using the Memo to represent the different alternatives makes the decision of whether or not to inline a CTE purely cost-based. Within the same query, some CTEs may be inlined, while others may not be inlined. Inlining also allows performing some optimizations such as pushing down predicates, distribution, sorting, etc. For example, consider the CTEProducer in Figure 9(a). The partial logical expression shown in Figure 9(b) shows a predicate on top of a CTEConsumer. Using the transformation rule that inlines the CTEConsumer would give us the partial expression in Figure 9(c). However, Orca by default tries to push down predicates as far as possible, which means that the expression in Figure 9(c) will eventually be transformed to the expression in Figure 9(d), which may significantly reduce the number of intermediate rows.

6.2 Avoiding Invalid Plans

In this section we describe the algorithm which checks plans and subplans for validity with respect to the configuration of CTEProducers and CTEConsumers. The algorithm fits within Orca's framework of passing down query requirements and deriving plan properties (cf. Section 3.2). The algorithm operates on the different alternative plans encoded by the Memo. However, for illustration purposes, we present here a recursive implementation of the algorithm which operates on a complete plan.

The main function is given in Algorithm 1. The input to this function is a plan (or a subplan), represented by its root node, and a list of CTE requirements. Each item in the list is a *CTESpec* object. The output of the function is also a list of CTESpec objects that represent the CTE configuration in the given subplan. Each CTESpec is a compact specification of an unresolved CTEProducer or CTEConsumer, comprised of : (i) the CTE id, and (ii) spec type: either 'p' for producer or 'c' for consumer. For simplicity, we will represent each CTESpec using a pair (id, type). For example (1, c) represents a CTESpec with id=1 and type *consumer*.

This function is called initially on the top node of the whole plan, and given an empty list of requirements. The function computes a CTESpec list for the current node (lines 1-2), then proceeds to process the child nodes, if any (lines 3-7). For each child, we compute a new set of CTE requirements depending on the requirement coming from the parent as well as the CTESpecs obtained from previous children (line 4). We then call the same function recursively for the child, passing the new requirement (line 5) and combine the returned CTESpecs with what has been returned from previous children (line 6). Finally the function checks whether or not the combined CTESpecs of the current node and its children satisfies the requirement coming from the parent. If not, then the current plan is invalid (lines 8-10). Otherwise, the combined CTESpec list is returned to the parent (line 11).

The algorithm uses a number of helper functions; The *ComputeCTESpec()* function is operator-specific, and is used to compute the *local* CTESpec representation for every different operator. The

Algorithm 1: DeriveCTEs

```
Input  : Node node, List reqParent
Output: List of CTESpecs
List specList;
specList.Add(node.ComputeCTESpec());
foreach child in node.children do
    List reqChild = Request(specList, reqParent);
    List specChild = DeriveCTEs(child, reqChild);
    Combine(specList, specChild);
end
if !Satisfies(specList, reqParent) then
    SignalInvalidPlan();
end
return specList;
```

implementation of ComputeCTESpec() for most operators returns an empty list. The only exceptions are for the CTEProducer and CTEConsumer operators; each of which returns a list with one CTESpec object having the CTEid of that operator.

The *Request()* function computes a new list of requirements for a given child node, taking into account the parent's requirements, and the CTESpecs returned from the previous children. The new requirements contain the following:

- Any CTESpec not required by the parent, but introduced by a previous child. For example, the Sequence node in Figure 8(d) receives an empty requirement list, since it is the root. Its first child reports (0, p). Therefore the requirement for the second child is (0, c)
- Any CTESpec required by the parent and not resolved by the previous children. For example, the join node in Figure 8(d) receives the requirement (0, c) from its parent. This requirement is not satisfied by the first child. Therefore it is passed down to the second child.

The *Combine()* function combines CTESpec lists obtained from the current node and its children to build CTESpec list for the whole subplan rooted by the current node. Combining multiple lists takes place as follows: If CTESpecs exist with the same id but different types, they cancel each other and are not part of the combined list. All remaining CTESpecs are copied into the combined list.

The *Satisfies()* function checks whether or not the CTE representation of the whole subplan satisfies the requirement passed down from the parent. This is accomplished by comparing the CTESpecs in both lists, and checking if they match.

Finally, the function *SignalInvalidPlan()* signals that the current plan being processed is invalid because the requirements are not satisfied. Hence, it cannot be considered as one of the possible execution plans for the given query.

As noted in the beginning of this section, our implementation of this algorithm operates on the Memo groups directly, not on the extracted plans. The requests and derived properties are passed from one Memo group to its children and vice versa. Signalling an invalid subplan simply means that this subplan is removed from the plan space, and not considered as part of any plan. In other words, we do not wait until all alternative plans are produced and then apply this algorithm, but rather apply it as part of property derivation to avoid invalid operator combinations.

6.3 Optimizations Across Consumers

Execution plans containing CTEs can be further optimized in multiple ways to improve execution performance. In this section, we outline some of these optimizations. These optimizations have to take into account all consumers of a given CTE, and cannot be applied locally to individual CTEConsumers.

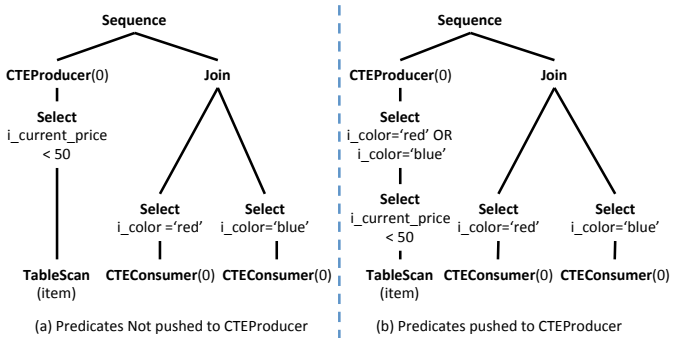


Figure 10: Pushing down predicates without inlining

6.3.1 Predicate Push-down

As explained in Section 6.1, inlining CTEs makes it possible to push down predicates, which reduces the number of intermediate rows. However, in Orca, we introduce a method to push down predicates even without inlining CTEs.

Example 5. Consider the following query:

```
WITH v as (SELECT i_brand, i_color FROM item
           WHERE i_current_price < 50)
SELECT * FROM v v1, v v2
WHERE v1.i_brand = v2.i_brand
AND v1.i_color = 'red'
AND v2.i_color = 'blue';
```

This query has two CTEConsumers, with a predicate on each one. Figure 10(a) shows a possible execution plan for the original query without CTE inlining. However, in this plan, the CTEProducer outputs tuples that are not needed by any of the CTEConsumers. We optimize this by forming a new predicate as the disjunction of all predicates on top of the CTEConsumers, and pushing that new predicate to the CTEProducer. This reduces the amount of data that needs to be materialized. We still need to apply the original predicates on top of the CTEConsumers to produce only the needed tuples. The optimized plan is given in Figure 10(b).

6.3.2 Always Inlining Single-use CTEs

In Orca, we use a heuristic where any CTE with a single consumer is automatically inlined. Consider the following example:

Example 6.

```
WITH v as (SELECT i_color FROM item
WHERE i_current_price < 50)
SELECT * FROM v
WHERE v.i_color = 'red';
```

This query has only one consumer of v. Whether we inline the CTE or not, the CTE expression is executed only once. However, when the CTE is not inlined, we also incur the added cost of materialization and reading back the materialized tuples. Therefore, it is always better to inline CTEs that are referenced only once.

6.3.3 Elimination of unused CTEs

Example 7.

```
WITH v as (SELECT i_color FROM item
WHERE i_current_price < 50)
SELECT * FROM item
WHERE item.i_color = 'red';
```

As an extension to the previous optimization, CTEs that are never referenced in the query can be removed altogether. In Example 7, v is defined but never referenced. No matter how complicated the definition of v is, it can be completely removed without affecting the result of the query. This can also be applied iteratively to queries with multiples CTEs.

Example 8. Consider the following query:

```
WITH v as (SELECT i_current_price p FROM item
           WHERE i_current_price < 50),
     w as (SELECT v1.p FROM v as v1, v as v2
           WHERE v1.p < v2.p)
SELECT * FROM item
WHERE item.i_color = 'red';
```

In this query, v is referenced twice, while w is never referenced. Therefore, we can eliminate the definition of w. However, by doing that, we remove the only references to v, which means we can also eliminate the definition of v.

7. CONTEXTUALIZED OPTIMIZATION

In this section, we discuss our novel contextualized optimization technique and give simple examples to highlight its impact. A key contribution of our work is the ability to optimize CTEs in different ways, depending on the contexts where they are used in different plans. Earlier efforts [11] have addressed this problem using a CTE re-optimization phase that exploits CTE properties discovered in previous regular optimization phases (cf. Section 2).

To the extent of our knowledge, our framework is the first proposal towards integrating CTE optimization at the core of the optimizer and avoiding full re-optimization. The cohesion of CTEs and other optimizations in one framework leads to an efficient and systematic process that eliminates redundant work. Efficient optimization often translates to better quality of the generated plans, since more resources can be devoted to expensive operations such as join ordering.

7.1 Enforcing Physical Properties

Orca optimizes candidate plans by processing *optimization requests* in Memo groups. An optimization request is a set of physical properties to be satisfied by a physical group expression. Required physical properties include sort order, distribution, rewindability, CTEs and data partitioning [2]. For clarity, we focus here on data distribution. Other properties are handled similarly.

For an incoming optimization request, each physical group expression passes corresponding requests to child Memo groups. These child requests depend on the incoming requirements as well as operator local requirements. During optimization, many identical requests may be received by the same group. Orca caches computed requests in a group hash table. An incoming request is computed only if it does not already exist in group hash table.

CTE optimization requires satisfying physical properties in different contexts, leading to potentially competing plan alternatives. We depict this process in Figure 11 using the query in Example 3.

7.1.1 Producer Context

A CTEProducer plan can be generated without considering where the CTEConsumers occur. To illustrate, in Figure 11(a), the Sequence operator requests ANY distribution from group 4, where the CTEProducer exists. This request can be satisfied by any plan generated from group 4. In this case, the derived distribution of the cheapest possible plan is Hashed(i_sk), which is the distribution of the item table. Next, the physical properties of this plan are attached to the request sent to group 1, from which the CTEConsumers descend. This allows the CTEConsumers to determine later whether the CTE plan satisfies the requirements at the CTEConsumers contexts, or property enforcers are needed to satisfy missing requirements.

The local requirements of the HashJoin operator in group 1 entail aligning child distributions based on the join condition (v1.i_brand = v2.i_brand). The goal is to collocate tuples that will join together on the same node. This is achieved by requesting Hashed(i_brand) distribution from both child groups 2 and 3. Note that CTE plan properties are still attached to these requests. Next, the CTEConsumers at groups 2 and 3 check whether the distribution of the CTE plan (Hashed(i_sk)) satisfies the requested distribution (Hashed(i_brand)) or not. In this case, it does not, and thus Redistribute (i_brand) enforcers are plugged in groups 2 and 3 to satisfy the required distributions.

Figure 11(a) shows a candidate plan resulting after the previous steps, where CTE results are redistributed twice on the same hash expression. Since the plan is unaware of the CTEConsumer contexts, such redundancy cannot be avoided. This may not be the best alternative due to wasting network and CPU resources[2]. We show next how more efficient plans can be generated for this example.

7.1.2 Consumer Context

Identical properties may be required in the contexts where CTEConsumers occur. These properties can be enforced in the CTEProducer expression to avoid repeated work.

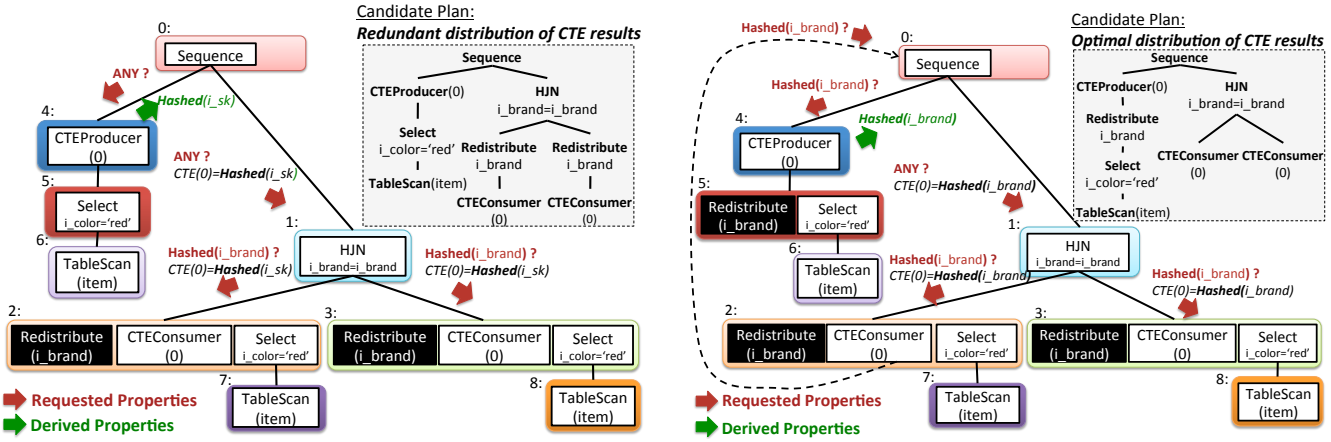
To illustrate, in Figure 11(b), the CTEConsumer operator sends an additional (Hashed(i_brand)) request (annotated with a dotted line) back to group 0, where the Sequence operator exists. In contrast to other optimization requests, this request does not arise from parent-child relationships in the Memo, but it is used to push CTEConsumers requirements into the CTEProducer plan. This is enabled by our framework, since each Sequence provides the optimization entry point in the Memo for its CTE. This optimization could result in avoiding property enforcement on the CTEConsumer side, as we show next.

Similar to the previous discussion, the request (Hashed(i_brand)) triggers a new sequence of optimization requests propagated across Memo groups. This results in adding a Redistribute (i_brand) operator to group 5 to enforce required distribution. When optimization reaches groups 2 and 3, the CTEConsumers recognize that the attached CTE plan properties satisfy Hashed(i_brand) distribution, and hence there is no need for property enforcing on the CTEConsumers side. Figure 11(b) shows a candidate plan, where CTE results are optimally distributed once before being shared by the CTEConsumers.

The plan alternatives where CTEConsumers are partially or fully inlined are still valid in our example. The optimizer estimates the cost of all these different alternatives in order to pick the best possible plan, as we discuss in Section 7.2.

It is also important to highlight that by integrating CTE optimization at the core of Orca, there is no need for multi-phase optimization, where CTEs are considered later after regular optimization is done. Additionally, by combining CTEs with other optimizations in one framework, comprehensive pruning of the search space is enabled. This is done by computing a lower bound on the cost of an optimization request, before fully computing it, and eliminating a request when a full plan with a smaller cost is already known. This allows cutting-off additional CTE optimizations early on. We omit further details due to space constraints.

[2]In other cases (e.g., when CTEConsumers have no distribution requirements), the plan generated from the Producer Context may be good enough.



(a) Enforcing properties on top of CTEConsumers

(b) Enforcing properties in CTEProducer expression

Figure 11: Enforcing physical properties for different plans for Example 3

7.2 Cost Estimation

Costing CTEs properly is crucial for optimization. The cost of the CTEProducer is the cost of executing the whole subtree under it, plus the cost of materializing the results to disk. The cost of a CTEConsumer is the cost of reading the rows from disk, so its cost is similar to the cost of a table scan. On the other hand, the cost of an inlined CTE expression is the full cost of executing that expression. For each CTEConsumer, the optimizer has two alternative plans:

1. A plan in which the CTE expression is inlined, and hence the full cost of executing the expression is incurred, and
2. A plan in which the CTE expression is not inlined, and hence we only incur the cost of reading the CTE outputs.

Comparing the cost of these two alternatives locally is incorrect, however, because the second alternative also implicitly assumes that somewhere else, a CTEProducer executes the expression and writes the results. The cost of the CTEProducer cannot be added to the reading cost of the CTEConsumer since the CTEProducer computes and writes data that can be read by multiple consumers, hence its overhead is amortized among these consumers.

As pointed out in [14], deciding on the best plan cannot take place locally for each CTEConsumer, but has to take into account all consumers of the same CTE, as well as the corresponding CTEProducer. Unlike the approach in [14], our approach does not require the additional work of computing the least common ancestor of all consumers of the same CTE, since this is already known; it is the Memo group which contains the corresponding CTEAnchor.

Since the CTEProducer is attached to the rest of the query via the Sequence, any plan alternative which includes the CTEProducer (e.g. Figure 3(b)) has its cost accounted for, regardless of the number of CTEConsumers. A plan which does not have a CTEProducer (e.g. Figure 3(a)) does not incur that extra cost. As a result, computing the cost and comparing the plans happens organically as the different plan alternatives are enumerated.

8. CTE-BASED OPTIMIZATIONS

We discuss how CTEs are implicitly generated by Orca as a way of optimizing queries that may not include explicitly defined CTEs. We show how CTEs can be used for optimizing some relational constructs such as distinct aggregates (Section 8.1), and for common subexpression elimination (Section 8.2).

8.1 CTE-Generating Transformations

In a number of scenarios, Orca employs transformation rules that implicitly generate CTEs during query optimization. For instance, while optimizing *window functions*, *full outer joins*, and *distinct aggregates*, Orca considers plan alternatives that employ CTEs. The following example describes one such scenario:

Example 9.

```
SELECT COUNT(DISTINCT cs_item_sk), AVG(DISTINCT cs_qty)
FROM catalog_sales WHERE cs_net_profit > 1000
```

The query in Example 9 computes two different distinct aggregates on `catalog_sales`. A possible MPP execution strategy of a single distinct aggregate requires input to be hash-distributed based on the aggregate column. This enables efficient identification of duplicates by sending identical values to the same node, and using multiple aggregation levels for de-duplication. However, when two (or more) different distinct aggregates are required (e.g., `COUNT(DISTINCT cs_item_sk)`, `AVG(DISTINCT cs_qty)`), this strategy becomes less efficient, since each aggregate entails a distribution on a different column.

In Orca, a rule that transforms different distinct aggregates into a join between CTEConsumers is used, so that we compute the input to different aggregates only once. The join serves as a means to concatenate the two aggregate values in one resulting tuple. Figure 12 illustrates the input and output of this transformation rule. Each CTEConsumer goes through a different Redistribute operator based on one aggregate column. This allows the MPP system nodes to perform distinct aggregate computation in parallel. For more than two distinct aggregates, the joins can be cascaded.

8.2 Common Subexpression Elimination

In addition to optimizing queries with explicit CTEs, our framework can also be used to optimize queries with common subexpressions that are not explicitly defined as CTEs. This is known as *common subexpression elimination*.

Example 10. Consider the following query:

```
SELECT *
FROM (SELECT i_brand, count(*) as b
      FROM item GROUP BY i_brand HAVING count(*) > 10) t1,
     (SELECT i_brand, count(*) as b
      FROM item GROUP BY i_brand HAVING count(*) > 20) t2
WHERE t1.i_brand <> t2.i_brand;
```

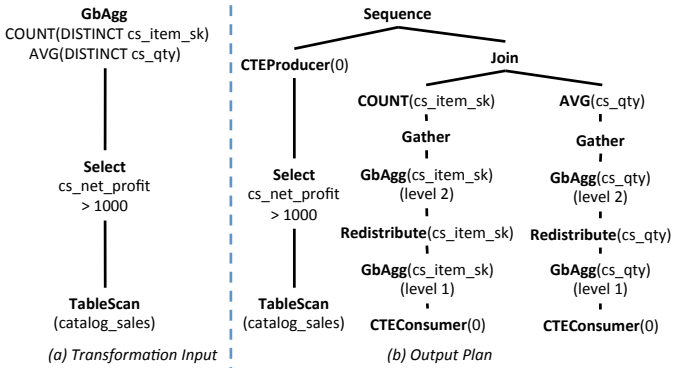


Figure 12: Generating CTEs for Multiple Distinct Aggregates

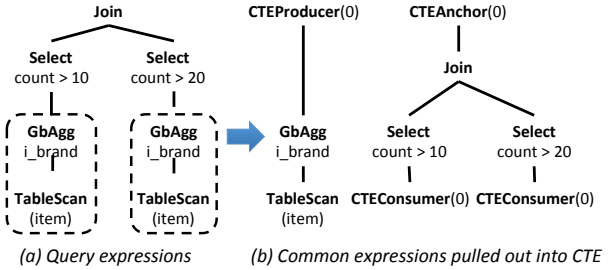


Figure 13: Common subexpression elimination

Algorithm 2: Common Subexpression Elimination

```
1  Algorithm EliminateCommonSubexpressions()
       Input  : Expression expr_in
       Output: Expression expr_out
2      M = DetectMatches(expr_in);
3      foreach (m ∈ M s.t. size(m) > 1) do
4          expr_p = any expression ∈ m;
5          p = create CTEProducer for expr_p ;
6          m.id = p.id;
7      end
8      expr_out = InsertCTEConsumers (expr_in, M);
9      foreach (m ∈ M s.t. m.used == true) do
10         l = LCA (m.consumers, expr_out) ;
11         expr_out = insert CTEAnchor(m.id) above l in expr_out
12     end
13     return expr_out ;
14 Procedure InsertCTEConsumers()
       Input  : Expression expr_in, SetOfMatches M
       Output: Expression expr_out
15     if (∃m ∈ M s.t. expr_in ∈ m ) then
16         m.used = true;
17         c = create CTEConsumer(m.id);
18         add c to m.consumers;
19         return c;
20     end
21     new_children = new set;
22     foreach (child ∈ expr_in.children) do
23         new_child = InsertCTEConsumers(child, M) ;
24         add new_child to new_children;
25     end
26     return new Expression(expr_in.op, new_children);
```

Figure 13(a) depicts the logical expression tree of this query. The query has a repeated common subexpression, outlined by the two dotted boxes. Algorithm 2 transforms an input logical expression $expr_{in}$ into an equivalent expression $expr_{out}$, where common subexpressions are replaced with CTEConsumers. The algorithm utilizes the function `DetectMatches()` to detect identical subexpressions (line 2). This function can be implemented in multiple ways. One example is using the *table signatures* method in [11, 14]. The output of `DetectMatches()` is a set M that is composed of groups of matching subexpressions in $expr_{in}$.

After M is computed, the algorithm visits groups in M with more than one member. For each such group m, a CTEProducer is created and the id of the created CTE is assigned to $m.id$ (lines 3-6). The algorithm then calls `InsertCTEConsumers()`, which recursively visits subexpressions, replacing each common subexpression with its corresponding CTEConsumer. Finally, a CTEAnchor is inserted above the *least common ancestor* (LCA) of each group of common subexpressions (lines 9-12). Figure 13(b) shows the output expression after processing the query in Example 10.

Note that this is only one possible algorithm. Other methods can be also used to identify similar (but non-identical) subexpressions. In this context, several view matching techniques can be leveraged by our framework, as we discuss in Section 2.

9. EXECUTION

In MPP databases, different parts of a query plan can execute in different processes, both within a single host, and across different hosts. In a shared-nothing MPP architecture such as GPDB, processes within the same host share a common filesystem, and processes in different hosts communicate through a network. The plans produced by Orca require CTEConsumers to read tuples from CTEProducers within the same host (both from the same process, and from a different process), but they *never* require reading from a CTEProducer on another host.

Before query execution starts, CTEConsumers are instantiated as SharedScan operators, and CTEProducers are instantiated as Materialize (Spool) or Sort nodes (if the CTE produces sorted results). A possible execution plan for the query in Example 3 is shown in Figure 14. The Broadcast operator manages data exchange between the shown two active processes (cf. Section 3.1).

Typically, a CTEProducer has multiple CTEConsumers executing both in the same process, as well as in other processes. The execution engine allows CTEConsumers to read tuples from CTEProducers in both of these scenarios. Additionally, when multiple processes are involved, the execution engine provides a synchronization mechanism to ensure the consumer can wait for the producer to have tuples. CTEProducer and CTEConsumers identify each other using the common CTEId. In addition, each CTEProducer is annotated with the number of consumers in the same process, as well as in different processes. These are used in the synchronization protocol between consumers and producers.

In GPDB, a process that contains a CTEConsumerexecutes a dependency check before it begins executing any of the operators assigned to it. Using a synchronization protocol, it waits for acknowledgement from the producer that all the tuples have been produced and are ready to be read. A process that does not contain any consumers executes normally. If any producers exist in a process, that producer notifies all the consumers once all the tuples are available. The synchronization protocol ensures that no notifications are lost, regardless of the order producers and consumers reach the synchronization point.

The results of a producer operator are explicitly materialized in a TupleStore, a data structure implementing an iterator over a set of tuples. The TupleStore for a CTEProducer can be stored in memory or on disk, depending on the size of the data, the amount of

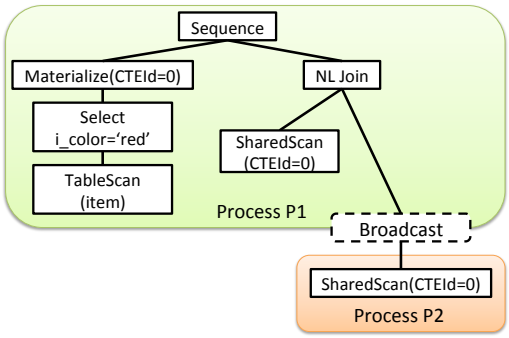


Figure 14: GPDB Execution Plan for Example 3

memory available and the plan requirements on the CTEProducer. When all the consumers of a particular CTEProducer are located in the same process, consumers can directly read the contents of the TupleStore from the producer. If the amount of data fits in the operator memory, the TupleStore is stored in memory, resulting in additional performance gains. If at least one consumer is located in a different process, then the TupleStore is stored on disk. The CTEConsumer receives the file name and the notification when the tuples are on disk, then proceeds to read the tuples from disk.

An additional optimization applies when the CTEProducer generates sorted tuples. If the operator on top of the producer is a Sort operator, explicit materialization of the results is avoided. The Sort operator already materializes its results in a TupleStore. This TupleStore is shared with the consumers directly.

The execution of plans with CTEs can be further improved by lazily executing the CTEProducer only when the first CTEConsumer requests it. If the CTEProducer and CTEConsumers are in the same process, a mechanism to jump between executing different parts of the plan is needed. If they reside in different processes, more coordination between processes is required, both for flow control and for efficiently sending tuples between processes on the same host. The CTEProducer must also be able to spill to disk when consumers request data at different rates. The cost of such plans cannot be estimated at optimization time, since it is based entirely on the execution flow; the cost estimates can vary depending on whether the CTEProducer executes in memory, spills to disk or is not executed altogether. We plan to investigate such improvements as part of future development.

10. EXPERIMENTS

In this section, we present our experimental evaluation. Section 10.1 outlines our experimental setup. Section 10.2 compares the performance of Orca-generated plans against plans generated by GPDB's legacy query optimizer, referred to as *Planner*, which always inlines CTEs. Section 10.3 outlines the importance of cost-based CTE inlining, as opposed to always inline CTEs or never inline CTEs. We do not experimentally compare our approach to the techniques proposed by other database systems. In general, this was infeasible because many of these systems are tightly coupled with their respective optimization/execution frameworks that are considerably different from GPDB and Orca. We highlight the similarities and differences between our approach and other approaches in Section 2.

10.1 Setup

The experiments were conducted on a cluster of eight nodes connected with 10Gbps Ethernet. Each node has dual Intel Xeon processors at 3.33GHz, 48GB RAM and twelve 600GB SAS drives in two RAID-5 groups. The operating system is Red Hat Enterprise Linux 5.5. We use the TPC-DS [13] benchmark with size 5TB. The workload consists of 48 queries – which are all TPC-DS queries containing CTEs.

10.2 Comparing Orca against the Planner

Setting	Execution Time
Orca	32,951.82 sec
Planner	57,176.04 sec

Table 1: Total execution time

Table 1 shows the total execution time of the entire workload using both Orca and Planner. Orca reduced the total execution time by 43%. Figure 15 shows the relative performance improvement of each query as a result of using Orca. The improvement is computed as a percentage of the execution time when using Planner, so that an improvement of 10% means a query finishes in 90% of the time, when using Orca. The X-axis is the execution time with Planner in logarithmic scale. One can see that across the board, Orca speeds up the execution time both for short-running and long-running queries. 80% of the queries exhibited performance improvement. This improvement is due to avoiding unnecessary inlining of CTEs, and thus avoiding re-executing common expressions, as well as the effective CTE optimizations discussed earlier.

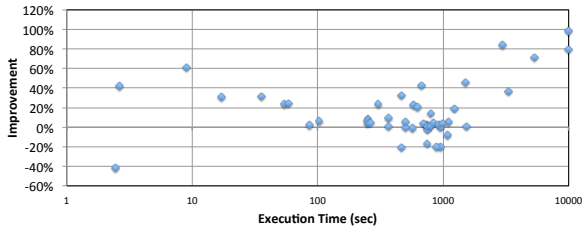


Figure 15: Performance of Orca vs. Planner

There are some instances, however, where Orca's plan is just as good as the plan generated by Planner. This is usually the case when the overhead of materializing and reading the CTE outputs from disk is roughly equal to the time saved from the multiple executions of the CTE expression in the Planner's plan. Lastly, for are some queries the performance degraded with Orca. We investigated these cases and found out that they are caused by Orca picking a suboptimal plan primarily due to imperfect tuning of cost model parameters and cardinality misestimation.

10.3 Cost-based Inlining

Next, we evaluate the effect of our cost-based CTE inlining method by executing our workload using the following settings:

1. Basic CTE optimization, where we disable CTE inlining as well as pushing down predicates into the CTEProducer.
2. Inlining is enforced for all CTEs.
3. Cost-based inlining, with all the optimizations enabled. This is Orca's default setting.

The goal of this experiment is to demonstrate that the *cost-based inlining* generates better plan by applying both principles (*pure inlining*, and *basic CTE optimization*) in a cost-based manner rather than applying only one of principles as a heuristic choice. Figure 16 shows the difference in performance among these settings.

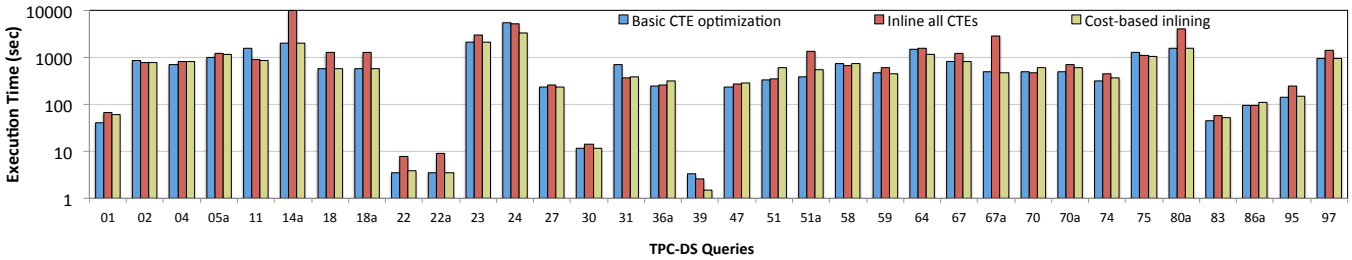


Figure 16: Performance Comparison: Basic CTE Optimization (No Inlining) vs. Inlining Only Approach vs Cost-based CTE Approach (uses inlining and CTE optimization techniques)

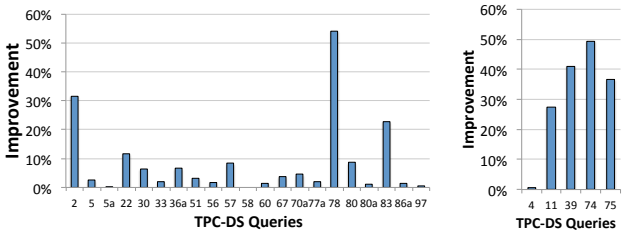


Figure 17: Effect of individual CTE optimizations: (a) inlining CTEs with a single consumer (b) pushing predicates down into CTE producer

The vertical axis represents the execution time in log scale. We observed that for 14 out of the 48 TPC-DS queries the difference in execution time between the three different settings was less than 10% and hence ignored in Figure 16. In the remaining 34 queries, we observed that if we picked inlining as a heuristic it can result on an average 44% performance regression and in the worst case a 4x regression as is the case for query 14a. A purely inlining CTE approach delivers notable performance gains only in scenarios where the inlined CTE is highly selective, namely queries 11 and 31 where the inlined subquery has a cardinality of a few hundred tuples. In these two cases, only using the *basic CTE optimization* can cause a 2x regression. The cost-based inlining approach captures the best of both worlds. Inlining is favored when there is only one consumer (as shown in Figure 17(a)), or the CTE is cheap enough to re-execute. Additionally, the cost-based approach favors no inlining when it is expensive to execute the CTE expression repeatedly. This hybrid approach along with the benefits of pushing predicates down into the CTE producer (as in Figure 17(b)) saves up to 55% of the execution time with negligible optimization overhead.

11. SUMMARY

This paper presents a comprehensive framework for the representation, optimization and execution of CTEs. Our work considerably extends the optimizer's infrastructure and addresses multiple challenges pertinent to distributed query processing in MPP systems. We demonstrate the efficiency of our techniques using standard decision support benchmark.

12. REFERENCES

[1] PostgreSQL. `http://www.postgresql.org`.

[2] L. Antova, A. El-Helw, M. A. Soliman, Z. Gu, M. Petropoulos, and F. Waas. Optimizing Queries over Partitioned Tables in MPP Systems. In *SIGMOD*, pages 373–384, 2014.

[3] C. Bear, A. Lamb, and N. Tran. The Vertica Database: SQL RDBMS for Managing Big Data. In *MBDS*, 2012.

[4] S. Bellamkonda, R. Ahmed, A. Witkowski, A. Amor, M. Zaït, and C. C. Lin. Enhanced Subquery Optimizations in Oracle. *PVLDB*, 2(2):1366–1377, 2009.

[5] L. Chang, Z. Wang, T. Ma, L. Jian, L. Ma, A. Goldshuv, L. Lonergan, J. Cohen, C. Welton, G. Sherry, and M. Bhandarkar. HAWQ: A Massively Parallel Processing SQL Engine in Hadoop. In *SIGMOD*, pages 1223–1234, 2014.

[6] J. Goldstein and P. Larson. Optimizing Queries sing Materialized Views: A Practical, Scalable Solution. In *SIGMOD*, pages 331–342, 2001.

[7] G. Graefe. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.*, 18(3), 1995.

[8] L. L. Perez and C. M. Jermaine. History-aware query optimization with materialized intermediate views. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 520–531. IEEE, 2014.

[9] Pivotal. Greenplum Database. `http://www.pivotal.io/big-data/pivotal-greenplum-database`, 2013.

[10] S. Shankar, R. Nehme, J. Aguilar-Saborit, A. Chung, M. Elhemali, A. Halverson, E. Robinson, M. S. Subramanian, D. DeWitt, and C. Galindo-Legaria. Query Optimization in Microsoft SQL Server PDW. In *SIGMOD*, pages 767–776, 2012.

[11] Y. N. Silva, P. Larson, and J. Zhou. Exploiting Common Subexpressions for Cloud Query Processing. In *ICDE*, pages 1337–1348, 2012.

[12] M. A. Soliman, L. Antova, V. Raghavan, A. El-Helw, Z. Gu, E. Shen, G. C. Caragea, C. Garcia-Alvarado, F. Rahman, M. Petropoulos, F. Waas, S. Narayanan, K. Krikellas, and R. Baldwin. Orca: A Modular Query Optimizer Architecture for Big Data. In *SIGMOD*, pages 337–348, 2014.

[13] TPC. TPC-DS Benchmark. `http://www.tpc.org/tpcds`.

[14] J. Zhou, P. Larson, J. C. Freytag, and W. Lehner. Efficient Exploitation of Similar Subexpressions for Query Processing. In *SIGMOD*, pages 533–544, 2007.

[15] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating Partitioning and Parallel Plans into the SCOPE Optimizer. In *ICDE*, pages 1060–1071, 2010.