User-Optimizer Communication using Abstract Plans in Sybase ASE

Mihnea Andrei
Sybase France
Paris
France
mandrei@sybase.com

Patrik Valduriez
LIP6 Laboratory University
Paris 6
France
Patrick.Valduriez@lip6.fr

Abstract

Query optimizers are error prone, due to both their nature and the increased search space that modern query processing requires them to manage. This paper introduces the Sybase Adaptive Server Enterprise (ASE) Abstract Plan (AP) language, a novel technology that puts together a set of proven techniques to palliate optimizer mistaken decisions. The AP language is a 2-way user-optimizer communication mechanism based on a physical level relational algebra. AP expressions are used both by the optimizer to describe the plan that it selected and by the user to direct the optimizer choices. APs are not textually part of the query. They are persistent objects stored in the system catalogs. APs yield important performance gains by eliminating all optimizer errors.

1. Introduction

Modern database systems use sophisticated query-processing techniques, both in terms of the rich set of relational operators and algorithms implementing them, and in terms of the wide space of legal plans that bind those operators together. As a consequence, the task of the optimizer has become very hard. Also, optimizers handle models of query execution – and there's a conceptual difference between models and reality. The best model in the world tautologically leaves parts of the reality uncovered. In trivial terms, for any real life optimizer, no matter how sophisticated, there will always be a query, a database state and a system state where the optimizer takes the wrong decision. There is not such a thing as the perfect optimizer. If we agree that making occasionally errors is in the nature of query optimizers, then we need to find a way to cope with that. Out of the many solutions for such scenarios, a pragmatic one is to provide external hints to the optimizer.

Abstract Plans (APs) is a technology that relies on the same language for both the user to constrain or direct the optimizer's choices and the optimizer to inform the user of its decisions. The APs are not part of the SQL text of the queries that they refer to, hence giving or modifying the AP of a query never requires any application program changes.

The AP language implements a physical level relational algebra. It does not provide a semantically complete description of the query. The optimizer can not ignore the parsed SQL query and build the QEP solely based on its AP. For instance, the predicates are not described. All optimizers will push down predicates as far as possible. Rather, an AP expression provides a description of the QEP, at the level where the optimizer takes decisions. The optimizer is then expected to create a QEP that is valid with respect to the SQL query and that complies with the AP. This AP description of a QEP is independent of any actual data structure implementing the QEP, hence the abstract nature of APs.

Let us introduce the AP language with an example. Consider the query:

```
select r1, s1 from R, S
       where r2 = 0
         and s2 = 100
         and r3 = s3
```

This query illustrates the join of 2 tables, with a search clause on each and a join clause. The optimizer could select for this query a QEP involving direct index access paths to each table and the Nested Loops Join (NLJ) algorithm. The following AP describes this QEP:

```
(nl_join
       (i_scan i_s2 S)
       (i_scan i_r3 R)
)
```

The user can then decide to force another plan that she estimates better, for instance directing the optimizer to keep the same join order but to use a Merge Join (MJ) and

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment

**Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001**

letting the optimizer choose the access methods. To achieve this, she will give to the optimizer the following AP:

```
(m_join
        (scan S)
        (scan R)
)
```

The AP technology puts together a set of proven techniques in an innovative manner. Most commercial RDBMSs already have means to influence the behavior of the optimizer. Some have syntax for hints that force the selection of the access methods and to some extent of the join algorithms and of other optimizer decisions. Some RDBMSs have optimization levels, which globally limit the set of techniques that the optimizer may use. All RDBMSs' optimizers have means to inform the user of their decisions and some of them use precisely an enriched relational algebra notation. Some RDBMSs have means to associate optimization hints with a query without any modification of the query text. But, to our best knowledge, no commercial RDBMS covers all major aspects of the AP technology.

This brief scenario gives the intuition of the AP technology. The rest of this paper will try to build compelling evidence of the advantages brought by the AP system.

The paper is organized as follows. Section 2 exposes the problem of dealing with optimizer errors. The underlying concepts of the AP language are described in Section 3. Section 4 introduces the syntax and semantics of the language. High level directions for the implementation of this feature are given in Section 5. Section 6 discusses related work. Section 7 concludes.

2. Problem definition

The first question that springs to mind is: is it a valid direction to work on optimizer hints, rather than focusing the efforts on enhancing the optimizers up to the level of quality where they are error-free? To answer, let us explore at a high level the optimizer technology and what drives them to make errors.

The typical examples of proven state of the art RDBMs query processing technology is the iterators based execution model implemented by Starburst [Loh88, HaaFLP89, Haa...90] and then by Volcano [Gra90, GraD93, Gra94, GraCMD94] and its industrial derivatives. The Volcano model is of a nice conceptual economy, as it only handles iterators. An *open*()/*next*()/*close*() interface allows any iterator to be the child of any other one. This encapsulation greatly simplifies the task of implementing and integrating novel query processing technology into the execution engine. An incomplete list of commercially available modern query processing technology (see the excellent overview in [Gra93]) contains sub-query flattening and decorrelation techniques [Kim82, Gav87, Day87, SesPC96], join indices and materialized views [Val87, GupM95, AgrCN2000], eager/lazy aggregation [ChaS96, YanL95, GupHQ95], parallel execution [OzsV96], rich set of inner/outer/semi join algorithms [MisE92], rich set of index structures, etc., etc.

Ironically, these query execution technology enhancements had an opposite impact on the optimizer. There's a long history of RDBMS optimizer technology research, starting with the seminal system R [SelACLP79]. An overview of optimizer research is out of the scope of this paper. Let us only cite 2 Ph.D. theses that are almost 20 years apart [Koo80, Pel97] and a recent overview [Cha98]. Despite a very active and fruitful research, while most execution engine limitations have been removed and a rich set of algorithms and combinations have become available to the optimizer, the expectations lain on the optimizer itself have continuously overcome its actual capacity. The optimizer is expected to find the best plan, or at least a decently good plan, but definitely to avoid any bad plan. In addition, it is supposed to be as fast as possible, which reduces to traversing the minimal portion of the search space.

Ideally from an optimizer user's perspective, the optimizer should directly guess the best plan without building any useless plan fragment. Cost-based optimization being a NP problem, this is clearly an unreachable ideal. On deterministic machines, the ideal optimizer would enumerate then execute all possible plans and state *post factum* "This *is* the cheapest plan!". Given that such an algorithm would be slower than the slowest plan (the slowest plan being one of the many enumerated and executed ones), it is equally clearly unacceptable. Even this caricatural implementation fails to meet the stated ideal optimizer requirement of always producing the best plan. First, enumerating all possible execution plans is a hard problem *per se*, similar to generating all theorems in a formal system. Then, the system load has an important impact on the relative merit of 2 plans. For instance, the sorting of a derived table is *O(ct)* in physical IOs when the whole tuples set fits in main memory, but gets up to *O(NlogN)* on the number of tuples *N*, when the same memory is shared by several queries and none of them can get enough for a memory resident sort. Now, real life shows that there is no guarantee that at run-time the system load would be the same as at optimization time. The honest *post factum* statement would be "This *was* the cheapest plan!", together with a disclaimer on any guarantee that a second run would behave precisely the same.

The clear mismatch in the above considerations is between an ideal optimizer and real life database usage scenarios. The expectation that the former can act in the latter is not realistic. This takes us back to real life optimizers.

Real life optimizers implement models of query execution. They are actually very close to the ideal optimizer, with only two differences. First, they try to reach the right tradeoff between skipping the maximum

number of bad plans and not missing the best plan, by using heuristic based and cost-based pruning. Second, instead of executing the plans in this restricted subset, they actually simulate their execution by using a cost model. But both differences bring a fair amount of risk.

The risk of pruning is indeed to miss some or all of the good plans. To avoid pruning is not a viable alternative. The huge search space of complex queries would timeout the search engine, pruning de facto the area still to be inspected. Some optimizers solve this by sampling through the search space, i.e. by either randomly or evenly inspecting a limited given number of plans. However, the pruning done by these algorithms also caries the risk of missing a good plan.

Furthermore, the most accurate the cost, the most effective the pruning. Now, the heaviest weight in the cost is the physical IO. And estimating the physical IO is not trivial. It actually relies on modeling the disk pages behavior in the buffer cache. Even for top-down search engines, logical and physical IO costing is still performed bottom-up, as one needs to know the algorithms first. Top-down costing (as in the Columbia optimizer, [Bil97, Xu98]) generally uses very conservative costs, with little pruning power. Even with bottom-up costing, the real execution physical IO behavior is hard to model. The physical IO is determined by the buffer cache that mainly uses a LRU policy. However, for the left deep NLJ trees, it's hard to cost accurately the physical IO of a sub-plan, as it's the innermost tables, that are not yet included in the sub-plan, who give the LRU behavior.

This overview of accurate costing and right pruning is not a complete tour of the hard problems met by the optimization technology. Let us briefly mention the complexity added by parallel query processing [HasFV96, LanVZ93], object extensions [LanVZ92], recursion [DusG97], concurrency and low level IO behavior [Moh92], host variables as search arguments, columns values correlation, etc. Dynamic query plans, parametric and dynamic optimization [GrW89, Ant93, ColG94, Gang98, Gra2000, BouFMV2000] are promising technologies used to reduce the optimizer errors. They are more accurate models, but still only models and can not guarantee an error less behavior.

The optimization is thus an error-prone technology. Although some errors of a specific optimizer are indeed due to its intrinsic weakness, it is in general impossible to have the perfect optimizer – the one that is error-free. It is indeed important to invest effort in enhancing the optimizer technology and attempt to eliminate all imperfections. However, this attempt will always hit scenarios where it fails. Coping with optimizer errors, rather than ignoring their fatality, is a valid parallel direction, from both a technological and a research perspective.

The problem that emerges from these considerations can be stated as follows: *create the technology that palliates to any optimizer error and that involves minimal user effort.*

The AP technology addresses this problem. APs give the user the means to describe the QEP that the optimizer should have created, but failed.

3. The *Abstract Plan* Language

To start with, let us go back to the query given in Chapter 1. Depending on the available indices, there are many possible legal plans for this query. The AP descriptions for some of them are listed below.

a. Use index i_s2(S.s2) to scan S as the outer table of a NLJ and index i_r3(R.r3) to scan R as the inner one. This is the AP created by the optimizer in Chapter 1.

```
(nl_join
        (i_scan i_s2 S)
        (i_scan i_r3 R)
)
```

Note that the index choice indicates that the search clause is used to position the scan on S and the join clause on R. The AP does not describe predicates placement.

b. Merge join R and S, using the indices that provide the ordering needed by the merge predicate. This AP is the full description of a QEP that's compatible with the AP given by the user in Chapter 1.

```
(m_join
        (i_scan i_r3 R)
        (i_scan i_s3 S)
)
```

Here, the search clauses filter the tuples but cannot limit the scanned pages, as they are not on the indexed attributes.

c. Merge join R and S, using the indices limiting the scan, then sorting to obtain the needed ordering.

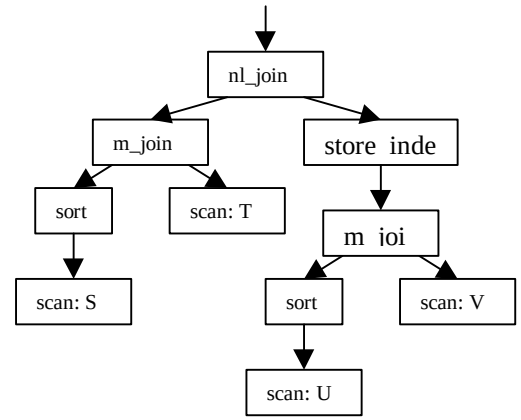
```
(m_join
        (sort
                (i_scan i_r2 R)
        )
        (sort
                (i_scan i_s2 S)
        )
)
```

d. Use the appropriate indices on both S and R to limit the scans with the search clauses, then join using nested loops by dynamically building an index (a.k.a. reformatting) on the inner derived table.

```
(nl_join
        (i_scan i_s2 S)
        (store_index
                (i_scan i_r2 R)
        )
)
```

These brief examples give a flavor of the strength and flexibility of the AP language. Let us now describe the underlying concepts of the AP technology.

The AP language is based on the derived table and stored tables concepts. Both concepts refer to their usual meaning in a RDBMS. But whereas an execution engine implements these concepts with QEP objects that will actually produce relational expression results and the optimizer implements them with objects that simulate (i.e.




```
(nl_join
        (m_join
                (sort
                        (scan S)
                )
                (scan T)
        )
        (store_index
                (m_join
                        (sort
                                (scan U)
                        )
                        (scan V)
                )
        )
)
```

Figure 1

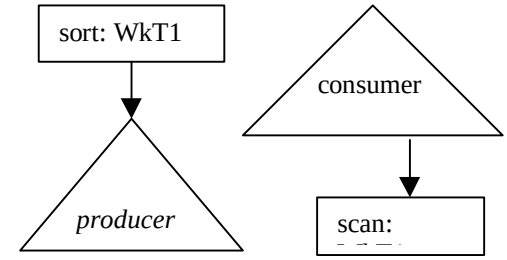
cost) the execution of such QEPs, the AP language uses them solely to describe a relational expression.

A stored table is a named and fully materialized collection of tuples, either a base table or a work table, and is described by an AP stored table. A derived table is the result of a QEP node. Both the node and its result are described by an AP operator.

The AP operators describe the relational algorithms handled by an optimizer, not the relational operators; they are at a physical and not at a logical level. For instance, for the typical implementations of the *join* operator, the AP contains the *nl_jo*in, *m_join* and *h_join* AP operators. The AP language can also give partial plans where some algorithms are not imposed and the choice is left open for the optimizer, so the AP language also contains the *join* AP operator.

The AP operators, as their QEP counterparts, have an arity, i.e. a number of arguments. These arguments are derived tables, i.e. other AP operators. Hence, the AP operator trees form a closed set with respect to composition.

A relational expression can be represented by a tree, having a node for each operator and an edge below that node for each operand of the operator. An AP is a textual representation of a relational expression tree. The




```
(sort WkT1
        ( ... the producer
                derived table ...
        )
)

( ... the consumer
        derived table
        ...
        (scan WkT1)
)
```

Figure 2

notation is inspired by [Frey87] and by the Lisp S-expressions.

Consider, for instance, 4 tables S-T-U-V and the relational expression that merge joins the 2 pairs of tables S-T and U-V sorting the outer, then nested loops joins these 2 results by reformatting the inner. The operator tree and the AP representing this expression are given in Figure 1. Note that the APs are isomorphic with relational expressions.

In an AP, worktables are hidden, whenever possible, as implementation details. For instance, the *sort* operator involves the usage of a worktable. If the worktable were exposed, it would involve 2 relational sub-trees (and 2 AP fragments), as shown in Figure 2. The AP of a query involving worktables would need means to associate several such AP fragments within the total AP of the query. The AP language approach was to hide such worktables as part of opaque AP operators (Figure 3).

Unfortunately, it is not always possible to hide the worktables. Take, for instance, shared relational sub-expressions, as self-joined views that are materialized. In such a case the QEPs contains two scan nodes that share the same worktable. A textual AP can not represent sharing without naming, and the most straightforward naming convention involves exposing the worktables. Hence, the AP language has the expressive power to name work tables and bind together AP fragments in a total AP, but this feature is used only for shared expressions.

The APs do not describe predicate placement, some non-ambiguous optimizer policy is assumed, as the deepest possible predicate pushdown. Likewise, the AP relies on the optimizer to place all scalar expression evaluation. The only expensive predicates and scalar

expressions currently described in the AP language are the subqueries:

```
select r1, s1 from r, s
where r3 = s3
  and r2 = (select t2 from t where t1 = r1)
(m_join
        (nest
                (i_scan i_r3 r)
                (subq
                        (i_scan i_t1 t)
                )
        )
        (i_scan i_s3 s)
)
```

The absence of scalars brings a lot of simplicity to the AP language, without reducing its expressive power: guiding an optimizer's search space traversal.

4. Syntax and Semantics

Let us use these underlying concepts to introduce the syntax and semantics of the AP language. The full AP language is not described in this paper. Such and undertaking would imply including most of the User Guide of a commercial product feature. Rather than being exhaustive at the risk of getting lost in language details, the focus will be lain on the relevance and conceptual integrity of a carefully selected subset of the AP language - the part that models joins and access methods. The grammar is given using Yacc [LevMB92] like rules.

The root non-terminal of the AP grammar, *abstract_plan*, is a derived table that describes the root relational operator of the QEP.

```
abstract_plan:
                derived_tab
;
```

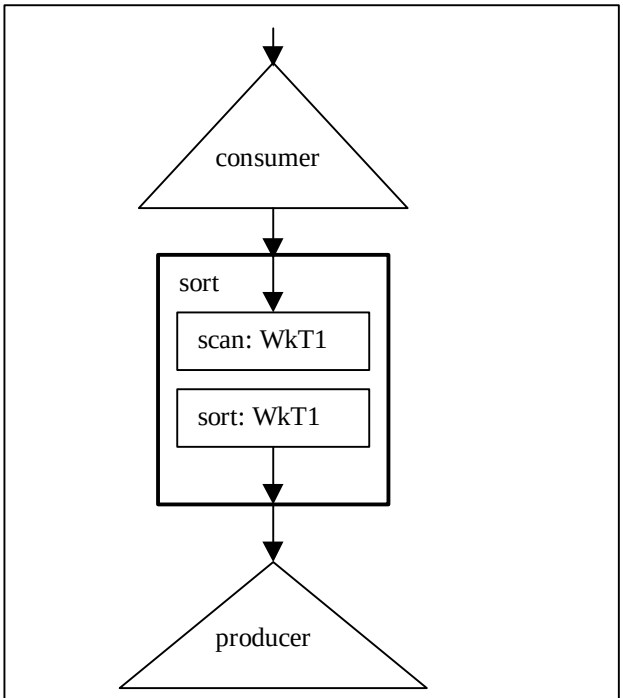
The *derived_tab* describes all the AP operators. Within the limits of this paper we will focus on (inner) joins, scans and enforcers.

```
derived_tab:    join
        |       sort
        |       xchg
        |       store_index
        |       scan
;
```

The *join* is a binary operator describing a join algorithm and the join structure. The term *join order* was introduced by the left deep tree limited QP engines. Modern QP engines can execute bushy tree plans. Some optimizers keep a preference for the left deep tree area of the search space, to decrease its size. But APs don't have this constraint, they describe any bushy tree, i.e. a *join structure*.

```
join:           nl_join
        |       m_join
        |       h_join
        |       any_join
;
nl_join: ( NL_JOIN derived_tab derived_tab )
;
m_join: ( M_JOIN derived_tab derived_tab )
;
h_join: ( H_JOIN derived_tab derived_tab )
;
any_join: ( JOIN derived_tab derived_tab )
;
```

The *any_join* syntax gives only a join structure, but leaves the optimizer free to choose the join algorithm.




```
(... the consumer
        derived table
        ...
        (sort
                ( ... the producer
                        derived table ...
                )
        )
)
```

Figure 3

Enforcers are unary operators that preserve unchanged in their output the relational contents of their input, but enforce on it a physical property, as ordering, partitioning, etc. The term *physical property* is found in [GraD87, GraCMD94, Gra95].

The optimizer could deduce the placement of some enforcers. This leads to the question: should the AP language describe them?

For instance, consider *ordering*, that is enforced by the *sort* operator. There's no doubt about the ordering available form a child, starting with the leaves (i.e. the access methods, where a B-tree index scan provides an ordering). Neither about the ordering needed by an algorithm (as a merge join, needing both children to be ordered on the merge predicate attributes). Hence, the optimizer could deduce the placement of the *sort* operator. But this would mean imposing a policy, as lazy enforcement – i.e. enforcing just below the point that needs the ordering. Other policies are possible, as some operators preserve the ordering of their children.

Hence, it is useful to describe with the AP language the placement of the *sort* enforcer.

```
sort:           ( SORT derived_tab )
```

```
;
```
The *sort* AP operator does not describe what attributes to sort on. Unlike the *sort* placement that allows several policies impacting the cost, for a given placement of a *sort* operator the optimizer can deduce the minimal set of attributes to sort on.

The *partitioning* is the physical property of a derived table that describes it being split up in several partitions (and, in a cluster environment, the node that contains each such partition), so that an independent clone of an operator can work on each in parallel. The Volcano model introduces in [Gra90, GraD93] the operator *Xchg* (read *exchange*) as the sole partitioning enforcer for all of the horizontal/vertical and SMP/cluster scenarios. The AP language implements it using the *xchg* operator.

```
xchg:          ( XCHG degree derived_tab )
;
```
The *degree* is an integer that gives the number of partitions. The optimizer could have deduced, as for the ordering, the minimal partitioning semantically needed at a node. However, the partitioning degree has an impact on the cost of a plan. It might be cheaper to split a derived table in more partitions than the minimum required by the semantics. Hence the presence of the degree in the AP *xchg* operator.

The *direct accessibility* is the physical property of an operator to provide a direct access to a subset of its tuples, as restricted by a predicate. The cost of a direct access operator is proportional to the size of its result set after the predicate was applied and not to the number of available tuples before the restriction.

The *store_index* AP operator is the enforcer that describes the materialization of its argument in an indexed work table (that is not exposed).

```
store_index:   ( STORE_INDEX derived_tab )
;
```

The leaves of the AP expressions are the scans, either one of the index direct access path or full table scan access path.

```
scan:          table_scan
       |       index_scan
       |       covered_index_scan
       |       any_scan
;
table_scan:            ( T_SCAN stored_tab )
;
index_scan: ( I_SACN index_desc stored_tab )
;
covered_index_scan: ( I_SCAN stored_index )
;
any_scan: ( SACN stored_tab )
;
```

As for joins, the *any_scan* syntax leaves the optimizer choose the access method.

The full implementation allows disambiguating between several occurrences of the same table in a query, either in the same FROM clause or in the different FROM clauses of the unions, views and subqueries contained therein. This is based on annotating, in the AP, the name of the stored table with its syntactic containment in views and/or subqueries, unions, etc. – according to its occurrences in the query, as in the example:

```
create view v(v1, v2) as
select * from t where t1 > 0

select * from t, v where t1 = v2
  and t2 = any (select t1 from t where t3 = 0)

(nl_join
       (m_join
              (i_scan i_t2 (table t (in
                                     (view v))))
              (i_scan i_t1 t)
       )
       (i_scan i_t1t3 (table t (in (subq 1))))
)
```

Here ends the join structure and access methods tour of the AP language, that we hoped brief but relevant.

5. Implementation

Before the AP interaction with the optimization process *per se*, let us start with the mechanism of associating a query with its corresponding AP. The AP of a query is not part of the SQL text. APs are stored in a persistent associative memory, where the lookup key is the SQL text of the query. Before the search space traversal, if the AP usage mode is active, a lookup is made and an AP text is potentially found – in which case it will influence the optimization process. Likewise, at the end of the optimization process, if the AP capture mode is active, then the AP describing the generated QEP is created – and stored, together with the SQL text of the query as an association key, in the associative memory. The persistent associative memory is implemented by an indexed system catalog. Hashing is used to speedup the search.

Let us focus now on the actual interaction between an AP and the internal workings of an optimizer. The main difficulty in implementing the AP technology is to prove that the AP describes a QEP that is valid, i.e. semantically equivalent to the query. Indeed, it is unacceptable to get different (hence wrong!) results from a query when an AP is used.

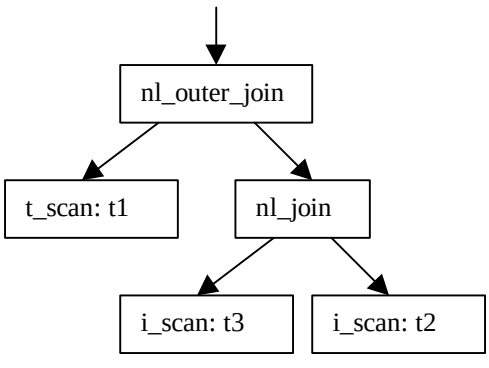
Let us give an example: inner joins permute with each other but do not always permute with outer or semi joins. Hence an AP could describe a join tree that is not compatible with the inner/outer/semi joins as they are in the SQL query.

For instance, the query below has an inner join as the inner term of an outer join:

```
select * from t1 left outer join
       (t2 inner join t3 on t2.c23 = t3.c32)
              on t1.c13 = t3.c31
```

A possible QEP for this query and the AP describing it are given in Figure 4. Now, the user could try to force a t3-t1-t2 join order with the following AP:

```
       (join
              (scan t3)
              (join
                     (scan t1)
                     (scan t2)
              )
       )
```




```
(nl_outer_join
        (t_scan t1)
        (nl_join
                (i_scan t3)
                (i_scan t2)
        )
)
```

Figure 4

With this AP, the optimizer is expected to find the appropriate algorithms for the *join* and *scan* operators, but preserve the join tree. However, none of the join AP operators can capture the outer join semantics of the query, as none of them has t2 and t3 in its inner sub-plan. Hence any QEP based on this AP would be wrong.

Actually, there are cases when such an AP would be correct. If referential integrity constraints guarantee that all t1 tuples qualify the ON clause, then the outer join becomes an inner join. But, to accept such a legal AP, the optimizer must be able to deduce that for this query and this database schema the outer join can be legally transformed to an inner join. And that reduces to building a relational algebra proof.

Hence the optimizer must build a proof. However, in its full generality this is a non-decidable problem. It involves comparing a canonical relational representation of the SQL query and the AP. The two expressions are semantically equivalent if one can be derived based on the other, within the limits of the relational algebra. This comes down to relational algebra theorem proving. But it is known that theorem proving is in general a non-decidable problem. It is thus impossible to prove the validity of all APs that have the same result set as the SQL query. To make the problem decidable, we must accept that some APs that are actually valid will be rejected by the system. For the AP implementation, as will be seen below, this statement will have a pragmatic reach. On a conceptual level, that this is the case for all non-decidable theorems: they are actually true, but that can't be proven.

More pragmatically, the AP implementation is supposed to accept only the APs describing QEPs that the optimizer *could have built*. Such QEPs might have been

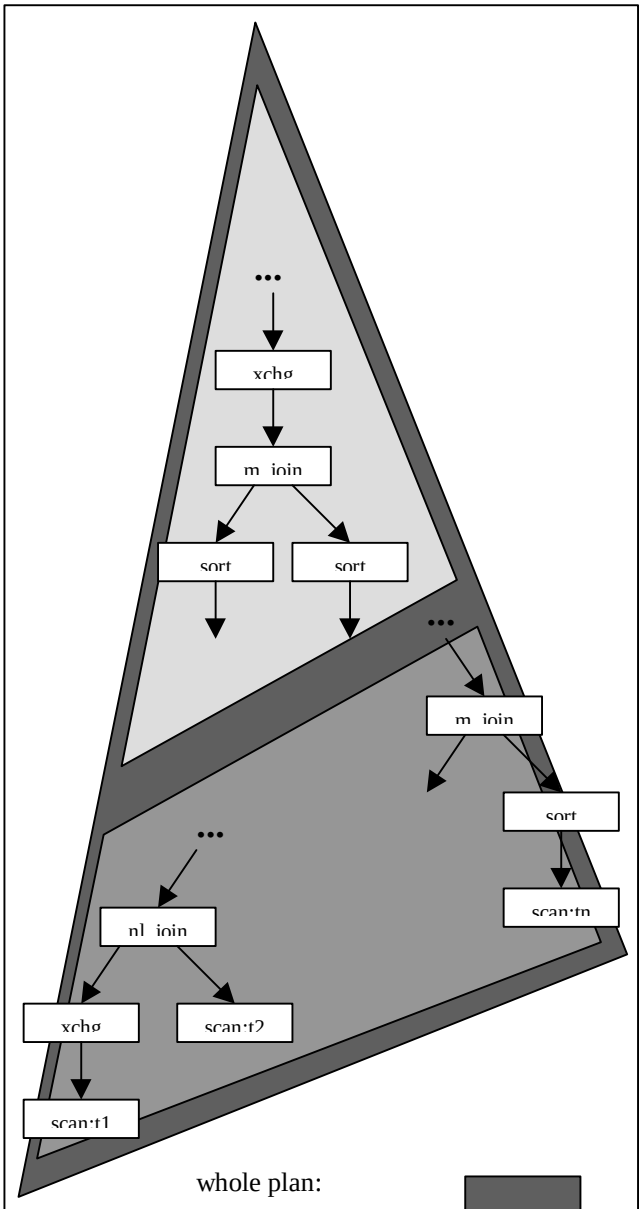


Figure 5

rejected due to costing or pruning, hence the need of an AP. But the optimizer had the logic to build that QEP, provided it was the cheapest one. An AP won't be able to instruct an optimizer to use, for instance, the magic set de-correlation strategy ([SesPC96]), if all that the optimizer knows are the enhanced Kim transformations ([Kim82, Gav87]). Note that this is far more restrictive that non-

decidable theorems. An optimizer can implement only a limited subset of all relational algebra transformations that are proven to be valid by some decidable theorem. All APs that can not be based on the implemented transformations will be rejected, even if a validity proof exists *in abstracto*. This conservative and restrictive choice makes the validity proof possible. It is also coherent with the primary purpose of the AP technology: palliate to the fatality of optimizer errors, rather than artificially enhance an optimizer's functionality.

This being said, let us give a high level view of the implementation of the AP technology within a modern Equivalence Class and plug-in search strategy based optimizer.

The term Equivalence Class (Eqc) is used as described in [GrCMD94]. An Eqc stands for a subset of the tables to be joined and contains the set of optimal sub-plans implementing the joins of that subset of tables.

A modern optimizer does a lot more than just left deep trees join order permutations and access path selection. It will attempt eager/lazy physical property enforcement, eager/lazy dynamic index creation, eager/lazy aggregation and delta-project, alternative subquery flattening and decorrelation techniques, basing access paths on complex AND/OR Boolean expressions, etc. When the optimizer transformation set gets richer, the task of proving the validity of all legal APs is more complex. Let us see how the validity proof is handled. The optimizer makes a strong distinction between policies and mechanisms. All transformations are implemented in the Eqc module, as mechanisms. These mechanisms are used by the search engine, whose search strategies implement the different policies. For instance, physical properties propagation, availability, need and enforcement is implemented in Eqc code, that is responsible to accept or reject a combination of parent and children join candidates, by checking the match between the available and needed properties. This code is independent of any costing decisions and makes no assumption on who will call its services; it just implements a pre-condition of a logical to physical transformation. When there is no AP, the search engine will use these services during the search space traversal – while attempting to join together the tables, within the plans it enumerates. When there is an AP, it replaces the invocation of the search engine for the fragments that it describes. Its validity proof is based on using the same Eqc methods to build plan fragments, as the search engine would use. The ability to build such a plan using the optimizer code that implements the known valid relational transformations is a constructive proof of the validity of the AP. If, in any Eqc, we obtain an empty set of sub-plans, then the optimizer failed to prove the validity of the AP. Such an AP is rejected.

A final note on AP performance enhancements and AP effectiveness. APs yield very important performance gains by eliminating all optimizer errors. Experience shows that small optimizer errors can propagate and cause huge performance degradations. Hence, APs can yield orders of magnitude of performance improvements. However, it is impossible to measure the performance enhancements of the AP technology itself, as they are actually equal to the performance losses caused by the optimizer. With the ideal perfect optimizer, there would be no performance enhancement at all, as the optimizer will always build the optimal QEP. Actually, such a perfect optimizer would not need the AP technology altogether. Alas, such a perfect optimizer does not exist. For a real life optimizer, we could define a metric over a selected set of queries, as the old TPC-D or the more recent TPC-H and TPC-R ones [TPC]. But such performance improvement numbers would actually measure the lack of quality of the optimizer rather than the quality of the AP technology.

The effectiveness of the AP can be measured, though. By effectiveness we mean the ability of the AP system to influence the optimizer to produce another QEP than the one that the optimizer would generate. To be able to measure that, we need an optimizer that makes errors. We have added a special optimizer status where the cost model will compute a cost that's opposite to the actual merit and the optimizer systematically generates the worse plan, making the wrong decisions at all levels.

This framework was used to test the AP system effectiveness in the following manner. The normal optimizer regression tests were used, that comes down to about 1200 queries exercising most optimizer features: 2 to 32 tables joins with several indices per table, aggregation, views, subqueries, unions, etc. A first run was made with the optimizer in its normal mode, capturing all the APs. The APs in this first set are describing the best QEPs for each query. Then, the error optimization level is enabled and a second run of the regression tests is made. During this second run, the optimizer would generate only bad QEPs. But it's not allowed to, as the set of best APs generated during the first run is used to force the QEPs. Also, during this second run all APs are again captured, in a second set of APs. If some optimizer decisions can't be constrained by the AP, and then the wrong costing model will generate a different QEP, hence a different AP. Hence, by comparing the two AP sets we get a measure of the AP system's effectiveness.

The measurement gave very encouraging results. Of the 1200 queries, only 15 had different APs – and the reason was outside the AP system itself. Those were the 15 queries that used random values in their search clauses. To the AP system, the two occurrences of such a query were two distinct queries, as their SQL texts differed.

6. Related Work

The authors are not aware of any related research, except for the effort of getting always better optimizers. To date, fallback solutions when the model

reaches its limits are not considered a research topic, but a pragmatic problem addressed by ad-hoc means in commercial RDBMSs.

To start with, the AP technology should not be confused with an "explain plan" feature. All commercial RDBMSs have means to describe the plans estimated by the optimizer, including costs, logical and physical properties, predicates placement, etc. Sybase ASE has such means, different from the AP system. Such an explain plan feature is meant to be as informative as possible for the user. The AP has an opposite aim. It's meant to be concise and support the bare minimum needed by the optimizer.

The means to influence the optimizer that are available in commercial RDBMSs are either optimization levels, or optimizer hints. The optimization levels have a coarse granularity and are far from the flexibility of the AP technology. Let us compare the latter with the two optimizer hints systems that come closest to it: the DB2 "reverse explain" and the Oracle "stored outlines".

The DB2 EXPLAIN system has, only for the System R based OS390/DB2 codeline, a reverse EXPLAIN feature. For all DB2 versions, the optimizer stores in an EXPLAIN table rows describing the selected plan. On the OS390/DB2 version, the optimizer can be instructed to use the plan captured in the EXPLAIN table, potentially modified by the user: the *reverse explain*.

The Oracle "stored outlines" are very similar to APs. They implement an optimizer hints language. The hints of a query are stored and retrieved, as the APs are, using the text of the SQL statement as association key.

The main difference between the AP technology and both the reverse explain and the stored outlines is the AP derived table concept. Both DB2 and Oracle features are based on stored tables. With each new table added to the join order, the hints can give the method to access that table, the algorithm to join these resulting tuples with the outer flow of tuples and whether any sort is needed at this level. This mechanism allows influencing only the portion of the plan from scan leaf nodes up to the first join nodes immediately above the scans, including any enforcers (as sorts) in between. Left deep trees can be fully described this way, as these are the only possible nodes. For bushy trees, the whole upper structure of the plan is out of reach, as illustrated in Figure 5. For instance, only the AP can force a complex bushy tree QEP, as the one in Figure 1. To our best knowledge, APs are the only commercially available technology to give optimizer hints, including enforcers, for any tree shape.

Note that the Starburst based UDB codeline of DB2 has an advanced EXPLAIN feature that can describe to the user any tree shape selected by the optimizer. However, it has no reverse EXPLAIN feature allowing the user to give optimizer hints.

7. Conclusion

The AP technology has proven its practical utility by providing a fallback solution for the cases when the selection of a sub-optimal plan was not considered as an optimizer bug but as an optimizer model limitation. The past experience in such cases was that trying to address an optimizer model limitation with a local fix, as changing a cost formula or a magic number, induces more problems than it solves.

The fundamental limitation of the AP technology is the mirror image of its strength: being static. When the data distribution changes, re-optimizing the queries at each invocation has the advantage that the QEPs adapt to the data changes – provided the optimizer takes the right decision. When the compilation of a QEP is based on a given AP, it does not follow the data changes, neither for the best nor for the worse. A citation from [Gra2000] is relevant here: "*Predictability versus risk is a more important dimension than fast versus slow, within the limits of common sense.*"

Bibliography

[AgrCN2000] Sanjay Agrawal, Surajit Chaudhuri, Vivek R. Narasayya: Automated Selection of Materialized Views and Indexes in SQL Databases. VLDB 2000: 496-505

[Ant93] Gennady Antoshenkov: Dynamic Query Optimization in Rdb/VMS. ICDE 1993: 538-547

[Bil97] Keith Billings: A TPC-D Model for Database Query Optimization in Cascades, M.S. Thesis, Portland State University, 1997

[BouFMV2000] Luc Bouganim, Françoise Fabret, C. Mohan, Patrick Valduriez: A Dynamic Query Processing Architecture for Data Integration Systems. IEEE Data Engineering Bulletin 23(2): 42-48 (2000)

[ChaS96] Surajit Chaudhuri, Kyuseok Shim: Optimizing Queries with Aggregate Views. EDBT 1996: 167-182

[Cha98] Surajit Chaudhuri: An Overview of Query Optimization in Relational Systems. PODS 1998: 34-43

[ColG94] Richard L. Cole, Goetz Graefe: Optimization of Dynamic Query Evaluation Plans. SIGMOD Conference 1994: 150-160

[Day87] Umeshwar Dayal: Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. VLDB 1987: 197-208

[DusG97] Oliver M. Duschka, Michael R. Genesereth: Answering Recursive Queries Using Views. PODS 1997: 109-116

[Frey87] Johann Christoph Freytag: A Rule-Based View of Query Optimization. SIGMOD Conference 1987: 173-180

[Gang98] Sumit Ganguly: Design and Analysis of Parametric Query Optimization Algorithms. VLDB 1998: 228-238

[GaV87]Richard A. Ganski, Harry K. T. Wong: Optimization of Nested SQL Queries Revisited. SIGMOD Conference 1987: 23-33

[GraD87] Goetz Graefe, David J. DeWitt: The EXODUS Optimizer Generator. SIGMOD Conference 1987: 160-172

[GraW89] Goetz Graefe, Karen Ward: Dynamic Query Evaluation Plans. SIGMOD Conference 1989: 358-366

[Gra90] Goetz Graefe: Encapsulation of Parallelism in the Volcano Query Processing System. SIGMOD Conference 1990: 102-111

[Gra93] Goetz Graefe: Query Evaluation Techniques for Large Databases. ACM Computing Surveys 25(2): 73-170 (1993)

[GraD93] Goetz Graefe, Diane L. Davison: Encapsulation of Parallelism and Architecture-Independence in Extensible Database Query Execution. TSE 19(8): 749-764 (1993)

[GraM93] Goetz Graefe, William J. McKenna: The Volcano Optimizer Generator: Extensibility and Efficient Search. ICDE 1993: 209-218

[Gra94] Goetz Graefe: Volcano - An Extensible and Parallel Query Evaluation System. TKDE 6(1): 120-135 (1994)

[GraCMD94] Goetz Graefe, Richard L. Cole, Diane L. Davison, William J. McKenna, Richard H. Wolniewicz: Extensible Query Optimization and Parallel Execution in Volcano. Query Processing for Advanced Database Systems, Dagstuhl 1991/1994: 305-335

[Gra95] Goetz Graefe: The Cascades Framework for Query Optimization. Data Engineering Bulletin 18(3): 19-29 (1995)

[Gra2000] Goetz Graefe: Dynamic Query Evaluation Plans: Some Course Corrections? IEEE Data Engineering Bulletin 23(2): 3-6 (2000)

[GupHQ95] Ashish Gupta, Venky Harinarayan, Dallan Quass: Aggregate-Query Processing in Data Warehousing Environments. VLDB 1995: 358-369

[GupM95] Ashish Gupta, Inderpal Singh Mumick: Maintenance of Materialized Views: Problems, Techniques, and Applications. Data Engineering Bulletin 18(2): 3-18 (1995)

[HaaFLP89] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, Hamid Pirahesh: Extensible Query Processing in Starburst. SIGMOD Conference 1989: 377-388

[Haa...90] Laura M. Haas, Walter Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce G. Lindsay, Hamid Pirahesh, Michael J. Carey, Eugene J. Shekita: Starburst Mid-Flight: As the Dust Clears. TKDE 2(1): 143-160 (1990)

[HasFV96] Waqar Hasan, Daniela Florescu, Patrick Valduriez: Open Issues in Parallel Query Optimization. SIGMOD Record 25(3): 28-33 (1996)

[Kim82] Won Kim: On Optimizing an SQL-like Nested Query. TODS 7(3): 443-469 (1982)

[Kooi80] Robert P. Kooi: The Optimization of Queries in Relational Databases. PhD Thesis, Case Western Reserve University, 1980

[LanVZ93] Rosana S. G. Lanzelotte, Patrick Valduriez, Mohamed Zaït: On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces. VLDB 1993: 493-504

[LanVZ92] Rosana S. G. Lanzelotte, Patrick Valduriez, Mohamed Zaït: Optimization of Object-Oriented Recursive Queries using Cost-Controlled Strategies. SIGMOD Conference 1992: 256-265

[LevMB92] John R. Levine, Tony Mason, Doug Brown: lex & yacc. O'Reilly & Associates 1992

[Loh88] Guy M. Lohman: Grammar-like Functional Rules for Representing Query Optimization Alternatives. SIGMOD Conference 1988: 18-27

[MisE92] Priti Mishra, Margaret H. Eich: Join Processing in Relational Databases. ACM Computing Surveys 24(1): 63-113 (1992)

[Moh92]C. Mohan: Interactions Between Query Optimization and Concurrency Control. RIDE-TQP 1992: 26-35

[OzsV96] M. Tamer Özsu, Patrick Valduriez: Distributed and Parallel Database Systems. ACM Computing Surveys 28(1): 125-128 (1996)

[Pel97] Arjan Pellenkoft: Probabilistic and Transformation based Query Optimization. PhD Thesis, University of Amsterdam, 1997

[SelACLP79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, Thomas G. Price: Access Path Selection in a Relational Database Management System. SIGMOD Conference 1979: 23-34

[SesPC96] Praveen Seshadri, Hamid Pirahesh, T. Y. Cliff Leung: Complex Query Decorrelation. ICDE 1996: 450-458

[TPC] Welcome to the TPC: http://www.tpc.org

[Val87] Patrick Valduriez: Join Indices. TODS 12(2): 218-246 (1987)

[YanL95] Weipeng P. Yan, Per-Åke Larson: Eager Aggregation and Lazy Aggregation. VLDB 1995: 345-357

[Xu98] Yongwen Xu: Efficiency in the Columbia Database Query Optimizer, M.S. Thesis, Portland State University, 1998