In 7th International Workshop on Database Programming Languages, 1999

# Optimization of Nested SQL Queries by Tableau Equivalence (Extended Abstract)

Vasilis Aggelis
*University of Patras*
aggelisv@otenet.gr

Stavros Cosmadakis
*University of Patras*
scosmada@cti.gr
Phone: 30-61-997867
Fax: 30-61-991650

**Abstract**

We present a new optimization method for nested SQL query blocks with aggregation operators. The method is derived from the theory of dependency implication. It unifies and generalizes previously proposed (seemingly unrelated) algorithms, and can incorporate general database dependencies given in the database schema.

We apply our method to query blocks with MAX, MIN aggregation operators. We obtain an algorithm which does not infer arithmetical constraints, and reduces optimization of such query blocks to the well-studied problem of tableau equivalence. We prove a *completeness* result for this algorithm: if two MAX, MIN blocks can be merged, the algorithm will detect this fact.

## 1 Introduction

The practical importance of optimizing queries in relational database systems has been recognized. Traditional systems optimize a given query by choosing among a set of execution plans, which include the possible orders of joins, the available join algorithms, and the data access methods that are used [SAC+79, JK84]. Such optimizers work well for the basic SELECT-FROM-WHERE queries of SQL [MS93]. However, they can perform poorly on *nested* SQL queries, which include, e.g., subqueries and views.

Since nesting of queries is a salient feature of the SQL language as used in practice, optimization of such queries was considered early on. One line of research has concentrated on extending the traditional "selection propagation" techniques to nested queries. In these approaches, traditional optimizers are enhanced with additional execution plans, where selection and join predicates are applied as early as possible [MFPR90a, MFPR90b, MPR90, LMS94]. Another line of work has proceeded in an orthogonal direction, introducing execution plans which correspond to alternative structures of nesting. In particular, these approaches consider the possibilities of merging query blocks, denesting queries, and commuting aggregation blocks with joins [Day87, GW87, Kim82, Mur92, PHH92, YL94, HG94].

In this paper we propose an approach which unifies and generalizes the approaches mentioned above. We apply the "selection propagation" idea to certain data dependencies that are implicit in aggregation blocks. Propagation of SQL predicates [MFPR90a, MFPR90b, MPR90, LMS94] is a special case of propagation of these dependencies. At the same time, propagating these

dependencies can produce execution plans with alternative nesting structure, as in [Day87, GW87, Kim82, Mur92, PHH92, YL94, HG94].

In addition to expressing in a common framework previously proposed query transformations which seemed unrelated, our approach incorporates naturally general data dependencies that may be given in the database schema. It extends transformations which commute joins with aggregation operators and merge query blocks [Day87, PHH92, YL94, HG94], in that it does not require adding tuple ids to the grouping addributes; and it can handle joins on aggregation attributes as well as on grouping attributes. Also, transformations which denest subqueries [GW87, Kim82, Mur92] only consider query blocks nested within each other, whereas our method does not depend on the order of nesting.

We illustrate our method by means of a small example. We consider the following database schema (of a hypothetical university database):

`ids`(Name, Idnum)
`enrolled`(Name, Idnum, Course)
`timetable`(Course, Hours)

The relation `ids` records the id numbers of students. The relation `enrolled` records the courses a student is enrolled in (and his/her id number); `timetable` records the number of hours a course is taught per week. These base relations do not contain duplicates.

The following dependencies are given in the database schema:

1. `enrolled`. Name,Idnum $\subseteq$ `ids`. Name,Idnum

2. `ids`: Name $\rightarrow$ Idnum

The first is an inclusion dependency (IND) stating that: every pair consisting of a student name and id number that appears in the `enrolled` relation, also appears in the `ids` relation. The second is a functional dependency (FD) stating that: student name is a key of the `ids` relation.

In Figure 1 we show a SQL definition for a view `maxhours` and a nested query $Q$.

The view `maxhours` gives, for each student, his id number; and the maximum number of hours of teaching (per week) of any of the courses he is enrolled in. The view `maxhours` is used to define the nested query $Q$, which gives, for each student, his id number and the maximum number of hours of teaching (per week) of any of the courses he is enrolled in; provided that there exist at least two courses which are taught for at least that many hours per week.

In Figure 2 we show the result of applying our optimization method to the nested query $Q$.

The query $Q'$ results by transforming $Q$ in the following ways.

First, the join with the `ids` relation in the main block of $Q$ is eliminated. This simplification is arrived at using the aggregation block of the view `maxhours`, and the dependencies of the schema. To justify the simplification we reason informally as follows. The join with the `enrolled` relation in `maxhours` is more restrictive than the join with the `ids` relation in the main block of $Q$, because of the IND 1. Also, the FD Name$\rightarrow$Idnum can be seen to hold for the `enrolled` relation, because of the FD 2 and the IND 1; consequently, the value of the Idnum attribute of $Q$ can be taken from the Idnum attribute of the `enrolled` relation, and thus from the Idnum attribute of `maxhours` (instead of the Idnum attribute of the `ids` relation).

The second optimization of $Q$ is that the subquery in the WHERE clause has been replaced by a view `countcourses`, which gives, for each number of hours some course is taught for, the number of courses that are taught for at least that many hours per week. Note that the common *nested*

*iteration* method of evaluating the subquery in $Q$ requires retrieving the `timetable` relation once for each tuple of the view `maxhours` referenced in the main block of $Q$. On the other hand, $Q'$ can be evaluated by single-level joins containing the join relations explicitly; this enables the optimizer to use a method such as *merge join* [SAC+79] to implement the joins, often at a great reduction cost over the nested iteration method [Kim82].

Observe also that the view `countcourses` contains the joins with the `enrolled` and `timetable` relations, appearing in the view `maxhours`. Including these joins makes the view `countcourses` safe, and produces a potentially cheaper execution plan, as it reduces the number of groups to be aggregated.

Optimization algorithms for nested SQL queries are often described as algebraic transformations, operating on a query graph which captures the relevant information in the query [MFPR90a, MFPR90b, MPR90, LMS94, Day87, GW87, Kim82, Mur92, PHH92, YL94, HG94]. In our method, we use the alternative *tableau* formalism that has been introduced in the context of conjunctive queries [AHV95, Ull89]. In Section 2 we sketch how this formalism is used to describe SQL queries.

In Section 3 we describe our optimization method; we use the chase procedure and the concept of tableau equivalence, which have been introduced for optimizing conjunctive queries in the presence of general data dependencies. One importance difference of SQL queries from conjunctive queries is the presence of duplicates in the result of a typical SQL query [IR95, CV93]. Our method optimizes correctly SQL queries where the number of duplicates is part of the semantics, and should not be altered by optimization[1].

We also describe in Section 3 how to fine-tune our method for the case of SQL queries with MAX, MIN operators. We obtain in this case an optimization algorithm which does not infer any arithmetical constraints.

In Section 4 we focus on the special case of *merging* of SQL query blocks with MAX, MIN operators. We show that, if such merging is possible, it will be discovered by our optimization method. Such completeness results can not hold for algebraic transformations of SQL queries: designing complete systems of algebraic transformations requires rather technical devices, having to do with the equality predicate [IL84].

In Section 5 we summarize our contribution, and point out some directions for further research.

## 2 SQL Queries as Tableaux

Tableaux are a declarative formalism which captures the SELECT-PROJECT-JOIN queries of the relational calculus. In this Section we describe (by example) a natural extension of tableaux which expresses SQL queries with nested blocks and aggregation operators. The tableaux we describe in this Extended Abstract express existential SQL queries, i.e., queries containing conditions which have to hold for *some* tuples in the database. An extension to queries with universal conditions – OUTER JOIN and null values – is described in the full paper.

For each query block we construct one tableau; subqueries or views within a query become separate tableaux. Figure 3 shows the tableaux for our example query in Figure 1.

A typical row of a tableau has the form R(x, y, ...), where R is the name of a base relation, a SQL predicate or a query block; and x, y, ... are variables local to the tableau, or constants.

The first row of a tableau gives the general form of a tuple in the result of the corresponding query block; it is called the *summary* row, and the variables it contains are called *distinguished*.

[1] The number of duplicates is irrelevant to the semantics of our example query $Q$.

`ids`(Name, Idnum)
`enrolled`(Name, Idnum, Course)
`timetable`(Course, Hours)

1. `enrolled`. Name,Idnum $\subseteq$ `ids`. Name,Idnum
2. `ids`: Name $\rightarrow$ Idnum

*V*: CREATE VIEW `maxhours`(Name, Idnum, Hours) AS
SELECT e.Name, e.Idnum, MAX(t.Hours)
FROM `enrolled` e, `timetable` t
WHERE e.Course = t.Course
GROUPBY e.Name, e.Idnum

*Q*: SELECT i.Name, i.Idnum, m.Hours
FROM `ids` i, `maxhours` m
WHERE m.Name = i.Name AND
$2 \leq$ ( SELECT COUNT (u.Course)
FROM `timetable` u
WHERE u.Hours $\geq$ m.Hours )

Figure 1: Example database schema and query

*Q'*: SELECT m.Name, m.Idnum, m.Hours
FROM `maxhours` m, `countcourses` k
WHERE m.Hours = k.Hours AND
$2 \leq$ k.Count

*W*: CREATE VIEW `countcourses`(Hours, Count) AS
SELECT t.Hours, COUNT(u.Course)
FROM `enrolled` e, `timetable` t, `timetable` u
WHERE e.Course = t.Course AND
u.Hours $\geq$ t.Hours
GROUPBY t.Hours

Figure 2: Optimized example query

The subsequent rows of the tableau give the general form of the tuples that have to be present in the base relations, and in the results of other query blocks; they typically contain additional variables, called *nondistinguished*.

Thus, for the tableau corresponding to the view `maxhours` the summary row is `maxhours`(n, p, hmax). The tuple (n, p, hmax) will be in the result of `maxhours` just in case the relation `enrolled` contains some tuple (n, p, c); and the relation `timetable` contains some tuple (c, h). Notice that c, h are nondistinguished variables. The last line of the tableau expresses aggregation and grouping: it states that, for each fixed n and p, hmax is the maximum possible value of h. A similar formulation of aggregation is described in [Klug82].

A tableau corresponding to a subquery contains non-local variables – they are local to the tableau obtained from the enclosing query block. These variables can be thought of as special constants of the tableau.

Thus, the tableau corresponding to the subquery in $Q$, *Qsubquery*, contains a non-local variable **H**, which is local to the tableau corresponding to $Q$.

It is straightforward (but lenghty) to give an algorithm which will convert a SQL query to a tableau representation; and vice versa. We defer the details to the full paper.

# 3 The Optimization Method

Optimization of tableaux (corresponding to conjunctive queries) has been studied extensively. The central notion is *equivalence*, i.e., finding a tableau which expresses the same query and can be evaluated more efficiently. The *chase* procedure is a general method to test equivalence of tableaux, in the presence of data dependencies [AHV95, Ull89].

Our method introduces, for each query tableau, an *embedded implicational dependency* (EID) [AHV95] stating that certain tuples exist and certain predicates hold in the database. In general, we can obtain such an EID by simply replicating the tableau.

Each query tableau is subsequently optimized using the dependencies of the schema and the EIDs introduced. The algorithm executes two passes (as in [LMS94]):

The first pass proceeds in a *bottom-up* way. Each tableau is optimized using the EIDs of the tableaux it contains. We start from the tableau which contain no subqueries or views, and finish with the top-level tableau.

In the second pass, each tableau is optimized using the EIDs of the tableaux it is contained in, in a *top-down* way.

In each pass, the optimization of each tableau consists of two distinct operations:

The first operation is to introduce new predicates; and to simplify the joins, by eliminating rows of the tableau.

The second operation is to replace subqueries by views (cf. the Introduction); it is done only during the second pass.

We illustrate the two operations by means of our running example.

Figure 4 shows the EID obtained from the view `maxhours`. It states that, for each tuple (n, p, hmax) in the result of `maxhours`, the relation `enrolled` contains a tuple (n, p, c); and the relation `timetable` contains a tuple (c, hmax), for some c. Notice that the EID is simpler than the tableau of `maxhours`. Such simplified EIDs can be used for query blocks with the MAX, MIN aggregation operators.

Introduction of new predicates and simplification of joins are done as follows.

The tableau is chased with the appropriate EIDs, and the dependencies of the schema. Figure 5 shows (in part) the result of applying this procedure to the tableau for $Q$. New rows are added

| | Name | Idnum | Course | Hours |
|---|---|---|---|---|
| `maxhours` | n | p | | hmax |
| `enrolled` | n | p | c | |
| `timetable` | | | c | h |
| hmax = MAX h (n, p) | | | | |

| | Name | Idnum | Hours |
|---|---|---|---|
| *Q* | n | p | **H** |
| `ids` | n | p | |
| `maxhours` | n | p' | **H** |
| | 1st | 2nd | |
| $\leq$ | 2 | C | |
| | Count | | |
| *Qsubquery* | C | | |

| | Count | |
|---|---|---|
| *Qsubquery* | c-count | |
| | Course | Hours |
| `timetable` | c | g |
| | 1st | 2nd |
| $\geq$ | g | **H** |
| c-count = COUNT c | | |

Figure 3: Tableaux for example query

| | Name | Idnum | Course | Hours |
|---|---|---|---|---|
| `maxhours` | n | p | | hmax |
| `enrolled` | n | p | c | |
| `timetable` | | | c | hmax |

Figure 4: EID from the view `maxhours`

to the tableau; they appear after the triple line. Chasing the second row of the original tableau with the EID obtained from `maxhours`, adds the first two of the new rows. Chasing the first of the new rows with the IND 1 of the schema adds the third new row.

The chase also adds to the tableau the SQL predicates appearing in the EIDs. In the case of the equality predicate, variables in the tableau are equated. In Figure 5, such equating happens by applying the FD 2 of the schema to the first and last rows; this equates p' with p.

To simplify the joins, the tableau resulting from the chase is minimized. This is done by examining the rows of the original tableau *not used in the chase*, and eliminating those which are covered by the tuples introduced by the chase. Note that it is not necessary for the chase itself to terminate; the tableau can still be minimized, as soon as a row as above is discovered.

Thus, the first row of the tableau in Figure 5 can be eliminated, because it is duplicated in the last row (recall that p' has been equated with p).

The final optimized tableau is obtained by dropping the rows that were introduced by the chase. In our example, this gives the tableau for Q' in Figure 6.

**Remark 1** *If the number of duplicates is part of the semantics of a query block, minimization of the corresponding tableau is omitted.*

Replacement of subqueries by views is done as follows.

The non-local variables of the tableau corresponding to a subquery are traced to the tableaux they are local to. The tuples containing those variables as local, are added to the subquery tableau. The resulting tableau is optimized by a method similar to the one used for the first operation.

Applying this operation to the tableau *Qsubquery* in Figure 3 (where **H** is a non-local variable) results in the tableau `countcourses` in Figure 6.

The correctness of our method is expressed in the following result.

**Theorem 2** *Suppose a query* Q' *is obtained by optimizing a query* Q.

*(i) On every database, the result of* Q' *contains exactly the same tuples as the result of* Q.

*(ii) If minimization is not used, each tuple is duplicated in the result of* Q' *the same number of times as in the result of* Q.

*(iii) If minimization is used, each tuple is duplicated in the result of* Q' *at most as many times as in the result of* Q.

We defer the proof of Theorem 2 to the full paper. The argument is a straightforward application of the properties of tableau chase and minimization, and of the results of [IR95, CV93].

| | Name | Idnum | Course | Hours |
|---|---|---|---|---|
| $Q$ | n | p | | **H** |
| ids | n | p | | |
| maxhours | n | p' | | **H** |
| | | ⋮ | | |
| enrolled | n | p' | c | |
| timetable | | | c | **H** |
| ids | n | p' | | |

Figure 5: Chase on the tableau of $Q$

| | Name | Idnum | Hours |
|---|---|---|---|
| $Q$' | n | p | **H** |
| maxhours | n | p | **H** |
| | Hours | Count | |
| countcourses | **H** | C | |
| | 1st | 2nd | |
| $\leq$ | 2 | C | |

| | Hours | Count | | |
|---|---|---|---|---|
| countcourses | **H** | c-count | | |
| | Name | Idnum | Course | Hours |
| enrolled | n | p' | c' | |
| timetable | | | c' | **H** |
| timetable | | | c | g |
| | 1st | 2nd | | |
| $\geq$ | g | **H** | | |
| | c-count = COUNT c | | | |

Figure 6: Tableaux for optimized example query

$Q_0$: SELECT i.Name, i.Idnum, m.Hours
FROM `ids` i, `maxhours` m
WHERE m.Name = i.Name

$Q_0'$: SELECT m.Name, m.Idnum, m.Hours
FROM `maxhours` m

Figure 7: Example of merging aggregation blocks

# 4 Completeness for Merging MAX, MIN Aggregation Blocks

It is not hard to see that nested SQL query blocks without aggregation can be merged. This is the Type-N and Type-J nesting considered in [Kim82]. Our optimization method can additionally merge query blocks where MAX, MIN operators are used in the inner block.

An example of such merging is shown in Figure 7; our running example is varied by omitting the last conjunct of the WHERE clause of $Q$, to obtain $Q_0$. The optimized block is $Q_0'$: essentially, $Q_0$ has been merged with the view `maxhours`.

There are cases where merging of MAX (MIN) query blocks can be shown to be impossible. Consider again the query $Q$ in our example. It is not hard to see that, by adding appropriately chosen tuples to the base relations, we can change the result of $Q$ to *empty*[2]. In contrast, this cannot happen for $Q_0'$, or its equivalent $Q_0$.

**Definition 3** *A query is* simple *if its result cannot be changed to* empty *by adding tuples to the database relations.*

**Proposition 4** *A SQL query defined by a single* MAX *block is simple.*

An analogous Proposition holds for SQL queries defined by a single MIN block.

By the above remarks, SQL query blocks cannot be merged into a single MAX block, unless the query defined is simple.

We can now state our completeness result.

**Theorem 5** *If a SQL query is simple, the optimization method transforms it into a single* MAX *block.*

An analogous result holds for transforming SQL queries into a single MIN block.

The proof is rather technical; we defer it to the full paper.

# 5 Conclusions

We have presented a general optimization method for nested SQL queries, which unifies several known approaches and at the same time extends them in several nontrivial ways. We have applied our method to the case of query blocks with MAX, MIN aggregation operators. For such queries, we have obtained an algorithm which avoids the complications of inferring arithmetical constraints [SRSS94, NSS98]; thus, it becomes possible to use algorithms for optimizing queries

[2] Consider the semantics of the last conjunct of the WHERE clause of $Q$.

without constraints [DBS90, CR97, ASU79a, ASU79b, JKlug84, CM77] to optimize nested SQL query blocks with MAX, MIN.

We believe our approach will be fruitfully applicable in other cases. A natural proposal is to apply it to aggregation operators which are known to be delicate to analyze, such as COUNT [Kim82, GW87, Mur92].

Finally, it should be possible to extend our approach to cover SQL queries with the ALL quantifier; and incorporate other optimization algorithms [RR98, SPL96] within our general framework.

## References


[AHV95] S. Abiteboul, R. Hull, V. Vianu. Foundations of databases. Addison-Wesley, 1995.

[ASU79a] A. Aho, Y. Sagiv, J. Ullman. Efficient optimization of a class of relational expressions. *ACM TODS* 4(4), 1979.

[ASU79b] A. Aho, Y. Sagiv, J. Ullman. Equivalence of relational expressions. *SIAM J. on Computing* 8(2), 1979.

[CM77] A. Chandra, P. Merlin. Optimal implementation of conjunctive queries in relational databases. In *STOC* 1977.

[CV93] S. Chaudhuri, M. Vardi. Optimization of real conjunctive queries. In *PODS 1993.*

[CR97] C. Chekuri, A. Rajaraman. Conjunctive query containment revisited. In *ICDT 1997.*

[Day87] U. Dayal. Of nests and trees: a unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *VLDB 1987.*

[DBS90] P. Dublish, J. Biskup, Y. Sagiv. Optimization of a subclass of conjunctive queries. In *ICDT 1990.*

[GW87] R. Ganski, H. Wong. Optimization of nested SQL queries revisited. In *SIGMOD 1987.*

[HG94] V. Harinarayan, A. Gupta. Generalized projections: a powerful query-optimization technique. Stanford University CS-TN-94-14, 1994.

[IL84] T. Imielinski, W. Lipski. The relational model of data and cylindrical algebras. *JCSS* 1984.

[IR95] E. Ioannidis, R. Ramakrishnan. Containment of conjunctive queries: beyond relations as sets. *ACM TODS* 20(3), 1995.

[JKlug84] D. Johnson, A. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *JCSS* 28, 1984.

[JK84] M. Jarke, J. Koch. Query optimization in database systems. *ACM Computing Surveys* 16(2), 1984.

[Kim82] W. Kim. On optimizing an SQL-like nested query. *ACM TODS* 7(3), 1982.

[Klug82] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *JACM* 29(3), 1982.

[LMS94] A. Levy, I. Mumick, Y. Sagiv. Query optimization by predicate move-around. In *VLDB 1994*.

[MFPR90a] I. Mumick, S. Finkelstein, H. Pirahesh, R. Ramakrishnan. Magic is relevant. In *SIGMOD 1990*.

[MFPR90b] I. Mumick, S. Finkelstein, H. Pirahesh, R. Ramakrishnan. Magic conditions. In *PODS 1990*.

[MPR90] I. Mumick, H. Pirahesh, R. Ramakrishnan. The magic of duplicates and aggregates. In *VLDB 1990*.

[MS93] J. Melton, A. Simon. Understanding the new SQL: a complete guide. Morgan Kaufmann, 1993.

[Mur92] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *VLDB 1992*.

[NSS98] W. Nutt, Y. Sagiv, S. Shurin. Deciding equivalences among aggregate queries. In *PODS 1998*.

[PHH92] H. Pirahesh, J. Hellerstein, W. Hasan. Extensible/rule based query rewrite optimization in Starburst. In *SIGMOD 1992*.

[RR98] J. Rao, K. A. Ross. Reusing invariants: a new strategy for correlated queries. In *SIGMOD 1998*.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, T. Price. Access path selection in a relational database management system. In *SIGMOD 1979*.

[SPL96] P. Seshadri, H. Pirahesh, T. Y. C. Leung. Complex Query Decorrelation. In *ICDE 96*.

[SRSS94] D. Srivastava, K. Ross, P. Stuckey, S. Sudarshan. Foundations of Aggregation Constraints. In *PPCP 1994*.

[Ull89] J. D. Ullman. Database and Knowledge-Base Systems, Vols I and II. Computer Science Press, 1989.

[YL94] W. Yan, P. Larson. Performing Group-By before Join. In *ICDE 1994*.