Rethinking Database Updates using a Multiple Assignment-based Approach

ELIZABETH HUDNOTT, JANE SINCLAIR, HUGH DARWEN
University of Warwick
Department of Computer Science
Gibbet Hill Road, Coventry
UNITED KINGDOM
lizzy.hudnott@gmail.com {jane, hugh}@dcs.warwick.ac.uk http://www.dcs.warwick.ac.uk

Abstract: We investigate the problems involved in efficient implementation of multiple assignment to database tables, as suggested by Date and Darwen in their *Third Manifesto* proposal for future database systems [10]. We explain the connection between assignment and the INSERT, DELETE and UPDATE operations and why multiple assignments executed simultaneously are preferable to deferred constraint checking. Our contributions are twofold. Firstly, we enable the user to specify updates either in terms of the changes needed to the existing state or as the final table contents directly, with no degradation in performance. Secondly, when multiple tables are updated SQL places the responsibility on the user to order the update statements correctly. Integrity constraints must either be preserved in the unnecessary intermediate states or else deferred. Multiple assignment accepts updates across the entire database simultaneously and makes the system responsible for scheduling them correctly. We present methods for a proposed implementation that can potentially exceed the performance of SQL DBMSs by employing parallelism and multi-query optimization.

Key–Words: Constraints, Multiple assignment, Multi-query optimization, Parallel updates, Query independence, Simultaneous assignment

1 Introduction

A *database state* is a collection of *relation variable names* (commonly known as “table names”) that denote a *relation value* (a “table” in SQL) consisting of zero or more tuples (loosely “rows”). Relation variables (or relvars for short) behave like conventional program variables in all respects except that they are accessible to other database programs (subject to security policy) and persist using non-volatile storage. Reading a relvar is equivalent to substituting its current value. For example, `SELECT * FROM R1 UNION SELECT * FROM R2` is equivalent to `SELECT * FROM VALUES ('Smith', 'London') AS R1 UNION SELECT * FROM VALUES ('Blake', 'Paris') AS R2` if `R1` and `R2` have the suggested values at runtime. DBMSs do not implement substitution directly because of performance concerns. Instead they capitalize on relations being composed from tuples by organizing them using memory layouts such as hash tables and B-trees that allow the DBMS to access only the tuples needed to answer the query. Similarly, a write operation is conceptually an *assignment* of a new relation value to a relation variable, and again a DBMS is free to implement any method that guarantees the same result as the conceptual model. Many programming languages provide shorthand for common assignments such as the familiar post increment operator ++ in C. DBMSs provide the INSERT, DELETE and UPDATE operators as shorthand for some assignments to relation variables. Without database integrity constraints any assignment can be emulated by a sequence of INSERTS, DELETES and UPDATES and in practice this is usually convenient. In fact, existing DBMSs only support the shorthand. However, the longhand <*relation var name*> `:=` <*query*> form is important for pedagogical purposes; for writing complex updates that are difficult to express using shorthand; and when the necessary sequence of shorthand assignments is prevented by integrity constraints.

Furthermore, the capability to assign to multiple relvars simultaneously is required for several reasons. The most familiar reason is to address situations where integrity constraints pre-

vent assignments from being executed sequentially that are nevertheless valid if considered together.

Multiple assignment offers performance advantages.

Parallel Execution: Assignments can be executed in parallel when they do not interfere with each other. In a distributed environment parallelism can be between assignments that are executed on the same or on different servers. For a discussion of the specific issues involved in multi-server parallism see reference [31].

Multi-Query Optimization: Commonality between assignments can be exploited by the optimizer.

Consolidated Updates: Multiple assignments to the same variable are reducible to a single assignment.

Multiple assignment is often used in formal specification languages [4] and some programming languages, but this work and Date and Darwen's *Third Manifesto* upon which it is based [10] is the first to apply multiple assignment to database tables. Some examples follow that demonstrate some of the key advantages of multiple assignment.

Example 1. *A company database contains details of employees and projects in three relvars:* `EMP`, `PROJ` *and* `EMP_PROJ`. *Every project must be have at least one employee and every employee-project attachment must refer to an existing project. Thus, when a project is created at least one employee must be simultaneously attached to it. The project details cannot be entered in* `PROJ` *before an employee is attached to it by insertion into* `EMP_PROJ` *and an employee cannot be attached to the project before its details are entered into* `PROJ`.

Example 2. *A variation occurs when using a database design technique called* horizontal decomposition *that is proposed by Darwen to avoid NULLs [9]. Suppose each project can use some resources, recorded in the* `RES` *and* `RES_USE` *relvars. Some resources are purchased specifically for one project and incur a cost to that project. Other resources are already owned by the company and do not incur any cost. A third group are needed for a successful project but have not been purchased yet so their cost is unknown. In an SQL database the second and third groups would be represented by NULL but NULLs complicate the relational model and can cause problems [11]. Under horizontal decomposition there are three additional relvars:* `COSTED_RES`{ `resourceID`, `projectID`, `cost` }, `NOCOST_RES`{ `resourceID` } *and* `UNCOSTED_RES`{`resourceID` }. *Some constraints are: (i) every costed resource must be a known resource, (ii) every resource without a cost must be a known resource, (iii) every uncosted resource must be a known resource, and (iv) every resource is exactly one of: costed, uncosted or without cost. Registering a new resource requires inserting a tuple into* `RES` *and simultaneously inserting another tuple into exactly one of* `COSTED_RES`, `NOCOST_RES` *or* `UNCOSTED_RES`.

Example 3. *A major project is completed and many employees involved are relocated to a subsidiary company, assigned new employee identification numbers and attached to projects in the subsidiary. Similar to Example 1, the new project attachments cannot be entered until the employee numbers are updated. The employee details cannot be updated until their previous project attachments are removed but the project attachments cannot be removed because that would leave employees with no current projects. Multiple simultaneous assignment is needed. The database constraints are guaranteed to hold prior to executing the assignment statement so the system can combine the foreign key declaration with the assignment statement [29] to reason that the constraints requiring revalidation are (a) the key constraints for both relvars, (b) that employees transferred to the subsidiary are attached to another project, and (c) that employees remaining with the parent company who were attached to the completed project have at least one other project attachment. SQL deferred constraint checking would incur an intermediate state where the database is inconsistent, making it impossible to reason from the database and the second update statement that only relocated employees and the completed project require checking. Thus the constraint would need revalidating in full.*

Example 4. *Suppose that an application involves simultaneous equations solved using the Jacobi method.*

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} \cdot x_j^k \right)$$

$x_1^k..x_n^k$ *are the values of attribute* `X` *for n tuples held in a relvar* `R` *after k iterations of a while loop that updates* `R` *and another relvar* `DIFF`. $x_1^0..x_n^0$ *are initial estimates of the mathematical unknowns* $x_1..x_n$ *that are successively refined by iteration.* a_{ij} *and* b_i *are constants, possibly held in other relvars.* `DIFF` *holds* $\{x_1^{k+1} - x_1^k, \ldots, x_n^{k+1} - x_n^k\}$ *for calculating the loop exit condition. The while loop contains two assignments, to* `R` *and* `DIFF` *which share a substantial common subexpression to calculate* $x_1^{k+1}..x_n^{k+1}$. *The update*

to `R` exhibits the Halloween Problem [22, pp.70–71] because updating one of the x_i tuples may alter its storage location and subsequently cause a naive update algorithm to update the same tuple twice within a single iteration k. Sequentially updating `DIFF` then `R` is possible but no known DBMS would recognize the expensive common subexpression. Consider an extension to the scenario. In weather forecasting $x_1..x_n$ represent some measurement (e.g. air pressure) measured at regular intervals on a map. Often a multi-resolution model is needed [8] though. For example, a national forecast for the UK needs fine grained simulation over the UK and coarse grained simulation of approaching weather systems from other parts of the world. Suppose that the global model and the regional model are held in separate relvars, `RG` and `RR`. Now $x_1..x_n$ are distributed between `RG` and `RR` and the update can only be performed in SQL by copying one of the relvars into a temporary table. Multiple assignment is a more convenient and less costly alternative.

Date gives further rationale and examples of multiple assignment in reference [12]. Multiple assignment is a comparable alternative to deferred constraint checking and provides the additional benefits of permitting dependency cycles such as the weather forecasting example, unhindered semantic optimization, and parallel execution.

2 Syntax & Semantics

A multiple assignment statement has the form:

```
V1 := E1, ... Vn := En;
```

`V1`..`Vn` are variables and `E1`..`En` are expressions (exclusively queries unless a `Vi` is a local program variable). If the same variable appears on the left-hand side of more than one assignment then the effect on that variable is the same as if those assignments were executed sequentially (apart from the possibility that sequential execution might raise a constraint violation when multiple assignment does not). However, assignments to different variables are simultaneous, that is, all of the expressions are calculated using the database state prior to any modifications [10, pp.177–180].

The structured operational semantics are given in Figure 1. Conceptually there are four steps. The first expands shorthand such as INSERT into `V := E` form. Shorthand is language specific and therefore not included in Figure 1. The second step handles multiple assignments to the same variable. The assignments are processed from left to right by the *replace* function. If an assignment `V := E2` is encountered subsequently to an assignment `V := E1` then `E1` is substituted wherever `V` occurs in `E2`. In this way any multiple assignment can be reduced to one where `V1`..`Vn` are unique. The third step introduces a temporary variable V' that is unused elsewhere for every variable V. The expressions are computed and stored in the corresponding temporary variables. The final step (described by the *msa* function) assigns the values of the temporary variables to their permanent counterparts and relinquishes the temporary variables.

$$\langle \texttt{V := E, } ACL\rangle, s \Rightarrow \langle replace((ACL), V, E)\rangle, s[V' \mapsto \mathbb{E}[\![\texttt{E}]\!]s] \quad (1)$$

$$\langle \texttt{V := E}\rangle, s \Rightarrow msa(s[V' \mapsto \mathbb{E}[\![\texttt{E}]\!]s]) \quad (2)$$

$$msa(s) = \{V \mapsto x | declared(V) \wedge assign(V, s) = x\} \quad (3)$$

$$assign(V, s) = \begin{cases} s(V') & \text{if } V' \in dom(s) \\ s(V) & \text{otherwise} \end{cases} \quad (4)$$

$$replace((\,), Vr, Er) = \epsilon \quad (5)$$

$$replace((\texttt{V := E, } ACL), Vr, Er) = \begin{cases} \texttt{V := E}[Er/Vr], ACL & \text{if } V = Vr \\ \texttt{V := E, } replace(ACL, Vr, Er) & \text{if } V \neq Vr \end{cases} \quad (6)$$

$\mathbb{E}[\![exp]\!]s$ denotes the result of evaluating expression exp in state s. ϵ denotes the empty string.

Figure 1: Formal semantics of multiple assignment

Our work is described in the context of the open source Ingres DBMS to provide a reference DBMS implementation to which multiple assignment could be added. The work is not dependent on any technical features unique to Ingres.

3 Optimizer Architecture

Our work is described in the context of Ingres to provide a reference onto which multiple assignment could be added, but the work is not dependent on any features unique to Ingres. Ingres uses a classic cost-based query optimizer. So statement execution proceeds in five phases: parsing, query rewrite, plan enumeration and selection, code generation, and finally scheduling

and execution. Some additional steps are needed to incorporate multiple assignment.

1. Parsing
2. Application of query rewrite rules
3. Dependency graph construction
4. Multi-query optimization, query plan enumeration and selection
5. Dependency cycle removal
6. Merging integrity constraint checks
7. Transitive dependency removal
8. Code generation
9. Scheduling and execution

4 Parsing

A multiple assignment statement enters the system via the parser where each expression is split into subqueries. A subquery in this context represents a select-project-join fragment as a list of result attribute specifications, a selection predicate and a range variable table. The term "subquery" in the SQL language specification has a different meaning. The range variable table records the relvars involved, including temporary tables for SQL subqueries. *Conceptually* the query begins by computing their Cartesian product. Join conditions are provided as part of the selection predicate. The overall query is a collection of subqueries to union, each formatted in a tree structure. Multiple assignment is part of Date and Darwen's **Tutorial D** language [10, Ch.5], which we are adding to Ingres as part of our work. **Tutorial D** permits more syntactic variations than are possible in SQL. For example, `R{X} WHERE X>5` is equivalent to `(R WHERE X>5){X}` (braces signify projection). The high degree of abstraction in Ingres's abstract syntax trees (ASTs) allows canonization of many **Tutorial D** variations during parsing, but not all. The parser expands INSERT, DELETE and UPDATE shorthand into longhand assignments and the statement is reduced to a single assignment per variable as previously discussed.

```
INSERT R S ⇒ R := R D_UNION S
DELETE R WHERE b ⇒ R := R WHERE NOT(b)
UPDATE R WHERE b (I := X, J := Y) ⇒
 R := SUBSTITUTE R WHEN b THEN (I := X,
     J := Y)
```

D_UNION is a variation of UNION that raises an error if the relations are not disjoint sets of tuples. SUBSTITUTE is not a **Tutorial D** keyword but it is useful to include an operator in the abstract syntax tree that directly corresponds to the single pass back-end operation of scanning a table and conditionally updating some attributes. In general there can be any number of substituted attributes. SUBSTITUTE is expressible in **Tutorial D** but the expression is not easily recognizable as a single pass operation.

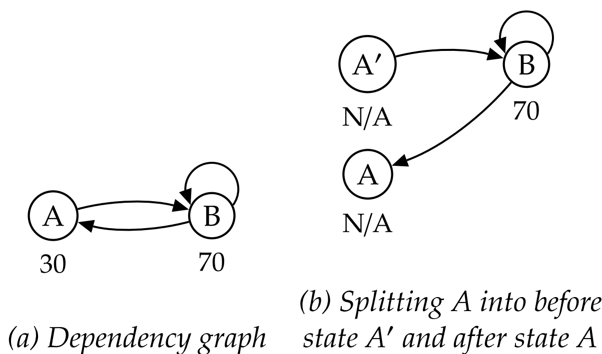
```
SUBSTITUTE R WHEN b THEN (I := X, J :=
    Y) ≡
(R WHERE NOT(b))
    UNION
    ((EXTEND (R WHERE b) ADD (X AS I', Y
         AS J')){ALL BUT I, J}
        RENAME (I' AS I'', J' AS J, I''
            AS I))
```

5 Query Rewrite

The conversion to an abstract syntax tree can only standardize variations that are easily detectable from localized pieces of syntax. More complex variations must be considered by traversing the tree after parsing is complete, which is performed by the query rewrite phase. **Tutorial D** queries have a less rigid structure than the SQL SELECT... FROM... WHERE... format. In particular, the placement of the UNION operator is more flexible. Therefore some rewrite rules not currently implemented in Ingres are suggested as additions in order to optimize query forms that would be unlikely to occur in SQL but are commonplace in **Tutorial D**. There is scope for adding further standardizations such as expanding parentheses in arithmetic expressions. We are considering these because multi-query optimization can make big performance gains when two equivalent subexpressions are identified. However, until the need for these rewrite rules is demonstrated in usability testing we rely on users writing their assignments in a consistent format for these more trivial variations.

6 Dependency Graphs

The examples illustrate read-write dependencies between the relvars, such as the local weather depending on the global weather. These dependencies are recorded using a *dependency graph*. A dependency graph is a directed graph with labelled vertices $G = (V, E, f)$. G has a vertex for each variable assigned, $V = \{\texttt{V1}, \ldots, \texttt{Vn}\}$ and an arc $Vi \rightarrow Vj$ if the assigned expression `Ei` might



(a) Dependency graph *(b) Splitting A into before state A′ and after state A*

Figure 2: Splitting a vertex

yield different results before and after the assignment `Vj := Ej` is performed. Informally, `Vi` must be executed before `Vj`. Formally, `Vi` is dependent on `Vj` if and only if `Ei` is ***not*** *conditionally independent of the update* `Vj := Ej`, that is, $Ei \not\equiv Ei[Ej/Vj]$ for some database that satisfies the database integrity constraints [27]. E is a superset of the dependencies because query dependence is undecidable in general. A dependency graph has a self-loop on `Vi` if `Ei` refers to `Vi`. A cycle involving two or more vertices indicates that there is no sequential execution order that assigns the correct values to all variables. Apart from a few special cases, one of the relvars must be version controlled to allow simultaneous access to both its pre and post assignment values, thus breaking the cycle (Figure 2) by splitting a vertex into two vertices. Some other techniques for resolving cycles in special cases are discussed in Section 8. Self-loops must be tracked but unlike true cycles many do not require special treatment, such as simple insertions. The remainder are instances of the well-known Halloween Problem, for which DBMSs already include handling mechanisms although version control is an alternative. The vertex labelling function f estimates the cost of the work needed to complete the update, produce version control information if needed and the overhead incurred from reading different versions.

Views must be replaced by their definitions to perform dependency analysis accurately. Similarly, if an expression invokes a user-defined operator then the operator definition must be examined to identify the relvars accessed in addition to the arguments. Greater precision can be obtained by simultaneously performing optimizations such as branching code elimination by substituting constant arguments in place of their formal parameters. Ingres does not currently perform any interprocedural optimization.

6.1 Arcs

Focussing specifically on deciding the arcs needed in the dependency graph, two query containment checks are needed to establish if a query is independent of an update, $Ei \equiv Ei[Ej/Vj]$: $Ei \subseteq Ei[Ej/Vj]$ and $Ei \supseteq Ei[Ej/Vj]$. Query containment checking is an established research problem [5, 20, 24, 26, 27] and in general the problem is NP-complete [7] so no solution to this difficult problem is attempted in the short space this paper permits. Any sound and complete algorithm documented elsewhere can be incorporated into the multiple assignment procedure. The approach that we recommend is the CQC method [15] because the authors show that it covers a broader class of queries than most previous methods, expands views and is at least as efficient as the other algorithms they survey. Also, CQC can take advantage that it is sufficient that the queries are equivalent only when the database conforms to its integrity constraints. The CQC approach handles the class of Datalog queries with negation (restriction, projection, join, rename, union and difference) and efficiently reasons about range restrictions for ordered data types. However, CQC cannot handle aggregation and is severely limited for recursive queries. Description logic approaches improve on the latter but cannot handle all select-project-join queries which are much more commonly than recursion.

Briefly, the CQC approach tests `Q1⊆Q2` by attempting to find a minimal example database where the query `Q1 MINUS Q2` returns some tuples. It sets out a proof using a tableau with five columns:

1. Goal (initialized to `Q1⊆Q2`)
2. Conditions to enforce (negations not yet processed, initialized to integrity constraints)
3. Example database constructed so far
4. Conditions to maintain (negations processed)
5. Literals used

The proof proceeds using proof rules that do the following transformations:

- Expand a view definition
- Make the database satisfy a non-negated term (1→3, 5) (A term moves from column 1 to column 3 with possible changes to column 5. Columns as numbered above.)
- Make a negated term into an additional constraint (1→2)

- Check a constraint (2→4, 5)
- Mark a constraint for rechecking (4→2)

When a query-update combination is not supported by the chosen independence checking method the system must make the conservative assumption that the query is dependent on the update if it refers to any updated projections of the updated relvar. This type of checking is similar to checking database schema dependencies and is much easier than the query independent of update problem [18].

6.2 Vertex Labels

The labelling function f gives the cost of the cheapest method of the two version control methods suggested. The primary suggested technique uses the differences between the old relvar value and the new value to rollback a page to obtain the older version no longer held in the database. The changes are obtained from the transaction log. This is a form of delta compression and it is already done by DBMSs that support multiple table versions for multiversion concurrency control (MVCC) schemes, such as PostgreSQL, Oracle and Microsoft SQL Server [32]. A variation is to store the differences in the database itself, as in Firebird [30]. Our implicit deletion technique is intended for use with this method. Either way, an existing MVCC system must be modified to allow the query engine to explicitly choose which version of a page the page manager supplies. Also, to calculate the dependency graph labels the optimizer must estimate the cost of completing the update and reconstructing the old version when needed. This is unnecessary for MVCC because there is no choice of which relvar to version.

When the delta compression technique is less suitable the alternative version control technique uses shadow copying to produce the new version. When the old and new values have little data in common, such as bulk unload operations, simple shadow copying can outperform the delta compression approach. Shadow copying can improve performance in the same cases even if no version control is needed because rows can be deleted without individual row deletion actions. The update cost for shadow copying is trivially calculated and there is no read overhead.

The costs are more complicated for delta compression. Conventional AIRES-style transaction logging is assumed [21] because it is much more common than other schemes such as Firebird's.

$$C_{Update} = C_{Search} + C_{Read} + \cdot C_{RLog} + C_{ULog} \qquad (7)$$

C_{Search} is the cost to use index lookups to find existing tuples to update or delete and to confirm the non-existence of tuples to insert. C_{Read} is the cost for reading the actual tuples to modify. C_{RLog} is the cost of writing the changed pages to disk as redo log entries and C_{ULog} is the cost to write the undo log entries. Let the assignment be `V := E`. C_{Search} + C_{Read} is the cost of querying the changed data, which is given by query plans for `V MATCHING E{K1,..., Kn}` (finding existing keys) and `E{K1,..., Kn} NOT MATCHING V` (confirming non-existence of new keys), where `K1,..., Kn` constitute a candidate key for `V`. Those query plans are subject to the restriction that a single table scan is the only operation allowed on `E` because materializing `E` as a temporary relvar would equate to shadow copying. The combined number of changed and added tuples is denoted by C.

Estimating the undo logging cost is straightforward under the simplifying assumption that each changed tuple produces one fixed-length log record. Let there be $N_{L/P}$ undo log records per page and let the system write log records N_{PL} pages at a time. T_{Seek} denotes the disk seek time and T_{Trans} denotes the block transfer time.

$$C_{ULog} = \left\lceil \frac{C}{N_{L/P} \cdot N_{PL}} \right\rceil \cdot T_{Seek} + \left\lceil \frac{C}{N_{L/P}} \right\rceil \cdot T_{Trans} \qquad (8)$$

The redo logging cost depends on the number of pages modified, which in turn depends on the locality of the affected tuples. For an unindexed or hash indexed relvar the distribution is essentially random. Let S denote the maximum number of tuples that fit into the storage currently allocated to `V` and $N_{T/P}$ denote the number of tuples that fit into a page. The minimum number of pages modified is $\lceil C/N_{T/P} \rceil$. The number of additional pages modified above the minimum follows a hypergeometric probability distribution. The probability of obtaining k affected tuples from a sample of n tuple slots follows the hypergeometric function $f(k, S, C, n)$. Considering the pages sequentially with n pages accessed so far, the probability that the 'next' page will be modified is the probability of obtaining some number of updated tuples r from $n + N_{T/P}$ tuple slots with at least one from the next sample of $N_{T/P}$ tuple slots. This is equivalent to sampling from two independent populations where the parameters of the second one depend on the sample taken from the first population.

$$
\begin{aligned}
&P(\text{modify next page}|\text{updated } r \text{ from } n) = \\
&\quad P(\text{update } r \text{ from } n)\times \\
&\qquad (1 - P(\text{update 0 from next } N_{T/P})) \\
&\quad = f(r, S, C, n) \times (1 - f(0, S - n, C - r, N_{T/P}))
\end{aligned} \tag{9}
$$

The unconditional probability is found by first summing over r and then summing over n in increments of $N_{T/P}$ (Equation 10). By numerical analysis the summation was discovered equivalent to a quartic equation in C (with small quantization errors) for fixed S and $N_{T/P}$. The summation function passes though $(1, 0)$, since when a single tuple is modified there cannot be more than one page modified in total, which is also the minimum number of pages. When $C > S - N_{T/P}$ the number of additional pages is zero because the minimum number of pages includes the whole relvar. By inspection there is one maximum at $(1/3S, 1/2 \cdot S/N_{T/P} + N_{T/P})$. The quartic curve can be approximated from these features by two quadratic curves and a plateau.

For relvars with physical data clustering (ISAM, BTree) the write cost is calculated using the optimizer's histogram estimation for querying the changed tuples to assign a seek time for each non-consecutive interval.

Estimating the overhead of reading the version of a relvar that is not stored in the database requires estimating how many updates will be made by overlapping transactions. Thus is the cost of rolling back a page to read the prior version if updates have been written back to database.

An in-memory table is used to record the number of transactions that make modifications to each relvar and the total number of transactions processed. When a transaction is initiated this counter is incremented and when it obtains a write lock the relvar's counter is incremented. If a transaction rolls back then the counters are decremented as the write locks are released. When a counter reaches its maximum possible value all of the counters are divided through by a small fixed constant.

Prior to a multiple assignment statement the DBMS computes the counter for each source relvar divided by the number of transactions counter to determine the proportion of recent transactions that have written to each relvar. Multiplying the result by the number of active read/write transactions gives an estimate of the length of the log record chain that must be interrogated to rollback a page.

7 Multi-Query Optimization

The expressions `E1..En` are queries that can be evaluated in any order. Normally a query optimizer optimizes each query independently as they arrive. However, multiple assignment inherently presents the queries `E1..En` to the system simultaneously. The sum of the cost of the best execution plan for each query considered independently may be more than the cost of the best plan that considers them together because of commonality between queries. Multi-query optimization (MQO) is more complicated than the common subexpression elimination that is often performed in a regular programming language compiler, or sometimes in hardware [28].

1. The query language is very expressive so semantic properties are compared rather than input text comparisons, hence the effort expended during parsing and query rewrite to put equivalent queries into a consistent format.

2. The only requirement is that two or more queries can execute faster if the result of another query is precomputed. For example `R{X, Y}` and `R{Y, Z}` have the common expression `R{X, Y, Z}` even though it is not a common *sub*expression.

3. The common expression must be temporarily stored somewhere. Relations are large data structures that do not usually fit into main memory so extracting a common expression incurs a cost that must be weighed against the cost saving. Modifying query plans to share a common expression is not always advantageous.

4. Reading from the precomputed common expression can become a bottleneck.

The first three problems are described well by Zhou et al. [34]. The forth problem can be addressed using a shared table scan technique [6]. As with query independence checking any available MQO algorithm can be used for multi-query optimization of multiple simultaneous assignments. However, we recommend the Zhou et al. approach because it has a proven implementation in a Microsoft SQL Server prototype that unlike many academic prototypes has a similar architecture to other industrial strength DBMSs. Our query rewrite rules attempt to move unions outward to maximize the size of the parse subtrees composed of only restriction, projection,

$$C_{RLogUnsorted} = \left(\frac{1}{N_{PL}} \cdot T_{Seek} + T_{Trans}\right) \times \left(\left\lceil \frac{C}{N_{T/P}} \right\rceil + \sum_{n=\left\lceil \frac{C}{N_{T/P}} \right\rceil \cdot N_{T/P}}^{S-N_{T/P},\, N_{T/P}} \sum_{r=\max(C-S+n, \left\lceil \frac{C}{N_{T/P}} \right\rceil + 1)}^{\min(C,n)} f(r,S,C,n)(1 - f(0, S-n, C-r, N_{T/B}))\right) \quad (10)$$

join and aggregation because these are the inputs to the Zhou et al. algorithm.

To briefly survey the issues involved in selecting an MQO algorithm: if there are n expressions that can share a common expression then the potential saving is $n-1$ times the cost of evaluating the common expression. The costs incurred are:

1. $1 \times$ the evaluation cost remains
2. $1 \times$ materializing the common expression
3. $n \times$ reading the common expression back in (can be reduced using shared scans)
4. Any additional operations to extract the requested data (e.g. further projections on `R{X, Y, Z}` above)

A significant problem is that adding MQO makes query optimization non-compositional. That is, the best plan for a query cannot necessarily be obtained by combining the best plans for its parts because those plans must take a share of the single evaluation and materialization costs. However, the number of participants that can benefit from a shared evaluation is unknown until a complete plan is produced. This problem is solved either by charging the shared costs to a common ancestor or by charging them at the point of use based upon a heuristic estimate of the number of consumers. Heuristics proposed include the number of uses in the best plan obtained without MQO [3], greedy algorithms, A* search and genetic algorithms [2]. The overall multi-query optimization algorithm can use text matching, an explicit search for compatible parse tree pieces (as in Zhou et al.), representing all possible query plans using a graph-based approach [3], costing the whole query plan by genetic algorithm, or by top-down analysis to devise the most beneficial view to partially cover the query expressions [23]. However, text matching is more limited than the other approaches and of the remaining approaches only a parse tree search can be retrofitted to a DBMS that uses standard bottom-up optimization.

E is derivable from V if and only if `E` can be computed by applying a sequence of in place update operations to `V`. In place update operations are: union, disjoint-union, intersection, semi-join, anti-join, substitution, restriction and transitive closure. If an operator has a second relation-typed argument then that is permitted to be an arbitrary relational expression. For example, `V := (V UNION R1) MINUS R2` can be transformed into `INSERT V R1; DELETE V R2;`, or more precisely internal versions of INSERT and DELETE that do not check integrity constraints. In place updates are usually considerably more efficient than (re-)materializing every tuple.

When a common expression `CE` is evaluated once for {`V1`, . . . , `Vm`} (without loss of generality) the dependency graph is updated as follows.

If some `Vk` ∈ {`V1`, . . . , `Vm`} is derivable from `CE` (Figure 3b):

1. Create `CE`:
   (a) Create a vertex for `CE`. Add outgoing arcs for the variables `CE` depends on.
   (b) Wherever there is an arc `Vi` → `Vk` for some `Vi` replace it by an arc `Vi` → `CE`.
2. Add usage arcs:
   (a) Add arcs from `CE` to each of {`V1`, . . . , `Vm`} \ {`Vk`}.
   (b) Add arcs from {`V1`, . . . , `Vm`} \ {`Vk`} to `Vk`.
3. Recheck dependencies: For each of {`V1`, . . . , `Vm`} check that the other outgoing dependencies are still necessary, ignoring variable references within `CE`. Delete the arcs for any unnecessary dependencies.

`Vk` is described as derived from `CE`.
Otherwise (Figure 3c):

1. Create `CE`: Create a vertex for `CE`. Add outgoing arcs for the variables `CE` depends on.
2. Add usage arcs: Add arcs from `CE` to each of {`V1`, . . . , `Vm`}.

3. Recheck dependencies: For each of {V1, . . . , Vm} check that the other outgoing dependencies are still necessary, ignoring variable references within CE. Delete the arcs for any unnecessary dependencies.

8 Dependency Cycles

If and only if the dependency graph does not contain any cycles (self-loops are permitted) then an ordering exists such that every assignment `Ei := Vi` is executed before any assignments to the relvars that Ei refers to. Cycles must be removed, for which four methods have been identified.

1. Version control
2. Catalog permutation
3. Further common expression factoring
4. Multi-query *de*optimization

Only version control is universally applicable, while the other methods handle special cases. Catalog permutation has negligible cost. The costs for version control were discussed in Section 6. The other methods use information calculated during multi-query optimization.

Finding the minimum cost of removing all cycles is more complicated than identifying the cheapest break-point on each cycle because a vertex can participate in multiple cycles. The problem, known as the minimum feedback vertex set problem, is NP-complete. However a reasonable approximation obtained quickly is sufficient for query optimization. Speed is especially important for small dependency graphs — the graph is constructed from a single multiple assignment statement so usually there are only a few vertices.

Using version control to split a vertex participating in multiple cycles breaks all of them for a single cost but splitting the vertex with the smallest weight is the cheapest way to break an individual cycle. Demetrescu and Finocchi [13] addressed this conflict by splitting the vertex with the smallest weight and subtracting the weight from the other vertices in the cycle. With the weights reduced they are more likely to be selected for breaking another cycle that is subsequently considered. If this occurs then two vertices have been split from the same cycle so a second phase recombines unnecessarily split vertices. The technique is extended similarly for the other cycle removal methods. Matrix multiplications dominate the asymptotic complexity and the approximation factor is the longest cycle length.

To split a vertex Vk into a pre-assignment value Vk and a post-assignment value Vk′ (Figure 2):

1. Create the vertex Vk′
2. For all Vj replace any arc Vk→Vj with Vk′ →Vj
3. For all Vi replace any arc Vi→Vk previously added by step 2 of multi-query optimization by Vi→Vk′.

Vk cannot be a candidate to split if it is derived from a common expression because an implausible directed path from the derived expression to the expression it derives from would remain. Other consumers can be split (e.g. B in Figure 3c) but if the common expression participates in the cycle then the cycle remains because of step 3 above and the split is unproductive.

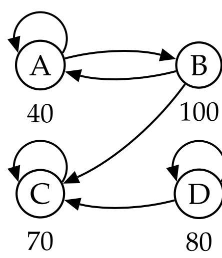
Catalog Permutation Although the multiple assignment `X := Y, Y := X;` has a dependency cycle it can be executed in constant time. The database catalog is updated so the identifier X points to the storage location previously allocated to Y and vice versa. The catalog update action is represented by a new vertex (Figure 4). In general, permutation is applicable whenever there is a cycle where none of the participating vertices are derived assignments from a common expression and no participants have self-loops. When a permutable cycle contains a subcycle then permuting the longest cycle breaks both cycles and gives greater concurrency. Vertices preceding the cycle can be placed before or after permutation (C in Figure 4).

Cycles and MQO Let $G = (V, E, f)$ be a dependency graph and $G' = (V', E', f')$ be obtained by performing multi-query optimization on G with a common expression CE. Let {C1, . . . , Cn} be the consumers of CE.

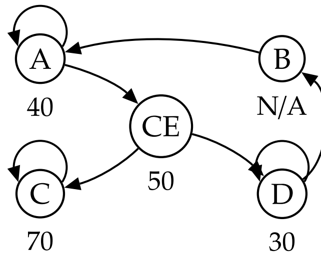
Theorem 5. *Let* Ck *be a derived assignment. If G has a cycle that contains* Ck *but no other consumers,* Ck → Vi → . . . → Vj → Ck, *then G' has a cycle that contains* CE.

Proof. (outline) By the rule for applying MQO to a dependency graph:

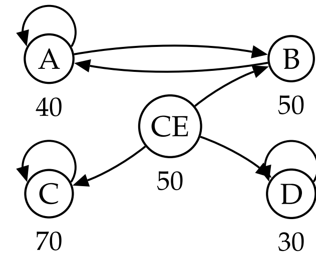
Case 1: CE depends on Vi. G' has a cycle CE → Vi → . . . → Vj → CE.



(a) Dependency graph

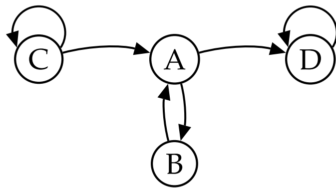


(b) `B` derived from `CE`

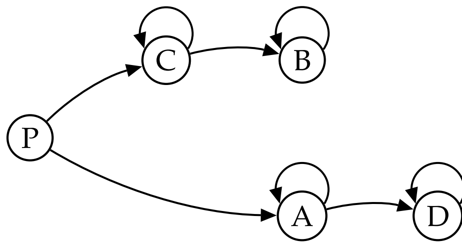


(c) `B` & `D` not derivable from `CE`

Figure 3: Multi-Query Optimization: `B` and `D` share a common expression involving `C`



(a) Dependency graph



(b) After catalog permutation

Figure 4: Removing the cycle $A \to B \to A$ by permuting the storage locations of A and B

Case 2: `Ck` depends on `Vi` outside of the common expression. G' has cycles `CE` $\to$ `Ci` $\to$ `Ck` $\to$ `Vi` $\to \ldots \to$ `Vj` $\to$ `CE` for each consumer `Ci` (except `Ck`).

Case 1, Case 2 or both always apply.

Theorem 6. *If there is no derived assignment and G has a cycle that contains exactly one consumer `Ck`, `Ck` $\to$ `Vi` $\to \ldots \to$ `Vj` $\to$ `Ck`, and `Ck` depends on `Vi` outside of the common expression then G' has a cycle that contains `Ck`.*

Proof. (outline) No arcs are removed by the rule for applying MQO to a dependency graph in this case. Therefore the cycle in G is preserved in G'.

Theorem 7. *If G' has a cycle `CE` $\to$ `Ci` $\to$ `Ck` $\to$ `Vi` $\to \ldots \to$ `Vj` $\to$ `CE` for consumers `Ci` and `Ck` where `Ck` is a derived assignment, then G has a cycle (or a self-loop if $\{Vi, \ldots, Vj\} = \emptyset$) `Ck` $\to$ `Vi` $\to \ldots \to$ `Vj` $\to$ `Ck`.*

Proof. omitted.

Theorem 8. *MQO with no derived assignment cannot introduce cycles.*

Proof. By definition of the MQO dependency graph rule the common expression has no incoming arcs.

MQO where one consumer is derived from the common expression adds two classes of arcs, either of which can introduce a cycle, though no introduced cycle includes both classes of arc added by the same optimization (Theorem 7): (i) arcs from common expressions to consumers, and (ii) arcs to the derived assignment from other consumers. Three levels of multi-query deoptimization are possible: (i) changing a derived assignment to a non-derived one (therefore making it not performed "in place"), (ii) removing the problematic consumer as a consumer, and (iii) total deoptimization.

9 Integrity Constraints

Integrity constraint checking introduces additional read dependencies to consider. An integrity constraint is a proposition that a given query does not produce any tuples. A vertex is added to the dependency graph for each integrity constraint and an arc is added from each relvar that a constraint is dependent upon to that constraint. Constraint vertices never have outgoing arcs so cycles are not possible. Defining constraints using queries allows constraints to participate in multi-query optimization.

10 Transitive Dependencies

Any transitive arcs remaining once the graph is acyclic are redundant. Redundant arcs can impair performance in a distributed environment because each arc requires a communication to the coordinator node. The transitive reduction of the dependency graph is found by first computing the transitive closure and applying Boolean matrix algebra operations [25].

11 Code Generation

Some of the syntactic constructs canonized during parsing suggest query plans that the canonical equivalents do not. For example, INSERT suggests a more efficient algorithm than an assignment where the right-hand side contains a D_UNION invocation. The canonical form provides a compact intermediate representation to simplify programming the optimizer. It also ensures equal performance irrespective of how the user states their update. However, an additional step is needed during code generation to select the best low-level operator such as "insert" or "general assign" to implement the intermediate representation. A similar selection process already happens for queries to choose the best join algorithm (for example hashed lookup versus sort-merge).

Some assignments can be executed in parallel while others must wait for dependent assignments to complete. The dependency graph is used to schedule the assignments. Each vertex has a counter that is initialized to the number of incoming arcs and protected by a mutex. There is a queue of pending jobs and a pool of worker threads. Threads execute an assignment and decrement the counters of all immediately proceeding vertices. A Counter reaches zero when all dependent assignments have been executed and the corresponding vertex becomes a pending job. Threads must frequently inspect a global flag that signals when a rollback is needed.

SQL does not support relvar assignment, only the shorthand INSERT, DELETE and UPDATE. An assignment `V := E` can be implemented in three ways.

1. Identifying the INSERT, DELETE and UPDATE operations that transform the current value of `V` into `E` and using them to update `V` in place. `E` is derived from `V`.

2. Generalized assignment. `E` must be fully computed, although only one tuple needs to be materialized at any time.

3. Proving that there are no deletions and optimizing the generalized assignment procedure accordingly.

4. Similarly, proving that all changes are deletions.

Derived assignments have already been discussed in connection with MQO. Here `E` derives directly from `V` whereas for MQO `E` derives from the common expression `CE`, which itself may or may not derive from `V` or another common expression. Thus an assignment can sometimes be separated into a derived part handled by in place operations and a non-derived part handled by generalized assignment. Query rewrite rules that attempt to maximize the part updated in place have been identified but are omitted here because of space constraints.

Generalized assignment has two phases and relies on a flag stored beside each row that indicates if it is new or has been updated, together with a field in the page header that records the transaction number of the last transaction that modified the page. This is a standard solution to the Halloween problem that is extended for generalized assignment. `E` is evaluated as a pipelined query and the tuples are fed to the assignment operator which either inserts or updates an existing row or "touches" an existing row (sets the flag without any update). The second phase performs a table scan on `V` and deletes any rows without the flag set. This can be eliminated by assuming *implicit deletion* if the DBMS implements multiversion concurrency control in the database rather than the transaction log. The flags are replaced by timestamps and if the row timestamp indicates an expired version them whenever the relvar is read then the row is known as having been deleted and is ignored. The second phase can also be eliminated if the query containment checking subroutine used to construct the dependency graph can be invoked to prove that `V := E` does not delete any tuples from `V`, that is `V{K1, ..., Kn}` ⊑ `E{K1, ..., Kn}`, where {`K1`, . . . ,`Kn`} is a key.

12 Related Work

Multiple simultaneous assignment was first described by Dijkstra [14] and its problem solving importance was subsequently discussed by Floyd

[16]. Some programming languages have multiple assignment, either of the kind discussed or often a very limited version where all participating variables must be assigned the same value. Notably the SQL UPDATE statement applies updates to multiple attributes simultaneously. However, attributes typically hold small amounts of data such as integers. We are not aware of any work that attempts a solution other than copying, despite modern SQL supporting user-defined types. Reference [19] is the only known work discussing efficiency. It describes a technique for minimizing the number of temporary variables used but the result is not necessarily minimal because the minimum feedback vertex set is not considered. "Multiple assignment" is also used to describe multiple assignments contained in the same statement that get executed sequentially, not simultaneously [33].

Multiple simultaneous assignment has equivalent semantics to an exclusive write PRAM. However prototype PRAM implementations do not support atomic operations on large data structures [1]. There are also similarities with the transaction scheduling problem [17]. However, some transaction schedules are always possible (non-interleaved, sequential) whereas dependency cycles are inherent in some multiple assignment statements.

13 Conclusion

We presented Date and Darwen's multiple assignment construct with a view to producing an efficient implementation suitable for large databases. A naive implementation prevents interference between assignments by copying the pre-assignment value of every variable into a temporary variable. We determine which assignments might interfere by using query containment checking to produce a dependency graph and avoid making complete copies by using version control techniques to record changes only. Also, SQL's lack of support for a general relvar assignment operator necessitated investigating efficient execution for an individual assignment. In addition to the challenges we considered performance related opportunities, including multi-query optimization, parallel execution and in particular how multi-query optimization interacts with resolving dependency cycles. We briefly discussed how integrity constraint enforcement procedures interact with multiple assignment, which we intend to conduct further work on. The next stage is to produce a quantified evaluation of our work. A working implementation of our techniques is not yet available because they involve substantial additions to many different components of the DBMS and our knowledge of the Ingres source code is limited. Generating appropriate performance tests requires careful consideration. Generic benchmark tests such as the TPC benchmarks are inappropriate because they do not exhibit the qualities that make some multiple assignments more challenging than others, such as dependency cycles. However, once suitable tests are constructed reliable numerical performance results could be produced prior to an implementation by applying our techniques manually and entering procedural SQL statements that mimic our code generation phase.

References:

[1] P. Bach, M. Braun, A. Formella, J. Friedrich, T. Grun, and C. Lichtenau. Building the 4 processor SB-PRAM prototype. In *Proceedings of the Thirtieth Hawaii International Conference on System Sciences (HICSS-30)*, volume 5, pages 14–23. IEEE Computer Society, 1997. 1060-3425.

[2] M. A. Bayir, A. Cosar, and I. H. Toroslu. Genetic algorithm for the multiple-query optimization problem. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 37(1):147–153, 2007.

[3] S. Bhobe, P. Roy, S. Seshadri, and S. Sudarshan. Efficient and extensible algorithms for multi query optimization. *SIGMOD Record*, 29(2):249–260, 2000.

[4] B. A. Brumar, E. M. Popa, I. Pah, and D. Chiribuca. Transition systems specified as a communication tool for e-learning. In N. E. Mastorakis, J. L. Mauri, H. Parsiani, K. L. Man, V. Mladenov, Z. Bojkovic, D. Simian, S. Kartalopoulos, A. Varonides, C. Udriste, E. Kindler, and S. Narayanan, editors, *Proceedings of the 12th WSEAS international conference on Computers*, pages 322–327, Stevens Point, Wisconsin, United States, 2008. World Scientific and Engineering Academy and Society (WSEAS).

[5] D. Calvanese, G. D. Giacomo, and M. Lenzerini. Conjunctive query containment and answering under description logic constraints. *ACM Transactions on Computational Logic (TOCL)*, 9(3):1–31, 2008. 1352590.

[6] Y. Cao, G. C. Das, C.-Y. Chan, and K.-L. Tan. Optimizing complex queries with multiple relation instances. In J. Wang, editor, *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD 2008)*, pages 525–538, New York, 2008. ACM. 1376671.

[7] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90, New York, 1977. ACM.

[8] J. Cote. Variable resolution techniques for weather prediction. *Meteorology and Atmospheric Physics*, 63(1):31–38, 1997.

[9] H. Darwen. How to handle missing information without using NULL, 2009, Jul 27 2005.

[10] H. Darwen and C. J. Date. *Databases, Types, and the Relational Model: The Third Manifesto*. Pearson Education, USA, 3rd edition, 2006.

[11] C. J. Date. *Oh No Not Nulls Again*, chapter 19. Addison Wesley, 1992.

[12] C. J. Date. *Date on Database: Writings 2000-2006*. Apress, USA, 2006. 1196397.

[13] C. Demetrescu and I. Finocchi. Combinatorial algorithms for feedback problems in directed graphs. *Information Processing Letters*, 86(3):129–136, 2003.

[14] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall series in automatic computation. Prentice-Hall, Englewood Cliffs ; London, 1976. 24cm.

[15] C. Farre, E. Teniente, and T. Urpi. Checking query containment with the CQC method. *Data and Knowledge Engineering*, 53(2):163–223, 2005.

[16] R. W. Floyd. The paradigms of programming. *Resonance*, 10(5):86–98, 2005.

[17] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation*. Prentice Hall, New Jersey, 2000.

[18] S. Gardikiotis and N. Malevris. Evolution and maintenance of database applications. *WSEAS Transactions on Computers*, 3(4):1081–1086, 2004.

[19] P. Grandi. Implementing (nondeterministic) parallel assignments. *Information Processing Letters*, 58(4):177–179, 1996.

[20] A. Gupta, Y. Sagiv, J. D. Ullman, and J. Widom. Constraint checking with partial information. In *Proceedings of the 13th ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 45–55, New York, USA, 1994. ACM.

[21] D. Haderle, B. Lindsay, C. Mohan, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992. 128770.

[22] J. M. Hellerstein and M. Stonebraker. *Readings in Database Systems*. MIT Press, Cambridge, Massachusetts, 4th edition, 2005. 2004113624 ill.; 28 cm. Includes bibliographical references.

[23] P. Kalnis and D. Papadias. Multi-query optimization for on-line analytical processing. *Information Systems*, 28(5):457–473, 2003.

[24] A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM (JACM)*, 35(1):146–160, 1988.

[25] D. C. Kozen. *The Design and Analysis of Algorithms*. Texts and monographs in computer science. Springer-Verlag, New York, 1992. 91036759. ill. ; 24 cm. Includes bibliographical references (p. 301-308) and index.

[26] M. Leclere and M.-L. Mugnier. Some algorithmic improvements for the containment problem of conjunctive queries with negation. In T. Schwentick and D. Suciu, editors, *Proceedings of the 11th International Conference on Database Theory (ICDT 2007)*, volume 4353/2006 of *Lecture Notes in Computer Science*, pages 404–418, Berlin / Heidelberg, 2006. Springer. 1611-3349.

[27] A. Y. Levy and Y. Sagiv. Queries independent of updates. In R. Agrawal, S. Baker, and D. A. Bell, editors, *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 171–181. Morgan Kaufmann, 1993. 672674.

[28] M. Martinez-Peiro, R. Gadea, R. Colom, V. Herrero, and F. Ballester. Multiplierless FPGA FIR filter design using a new signed common subexpression algorithm. In V. V. Kluev and N. E. Mastorakis, editors, *Proceedings of the WSEAS International Conference on Speech, Signal and Image Processing (SSIP 2001)*, pages 2061–2065. World Scientific and Engineering Academy and Society (WSEAS), 2001.

[29] G. N. Paulley. *Exploiting Functional Dependence in Query Optimization*. PhD thesis, Univerity of Waterloo, 2000.

[30] R. Rokytskyy. A not-so-very technical discussion of multi version concurrency control, 2008, Nov 3 2005.

[31] U. Shanker, M. Misra, and A. Sarje. Priority

assignment heuristic to cohorts executing in parallel. In N. E. Mastorakis, editor, *Proceedings of the 9th WSEAS International Conference on Computers*, pages 1–6, Stevens Point, Wisconsin, United States, 2005. World Scientific and Engineering Academy and Society (WSEAS). 1369714.

[32] R. Strohm. *Data Concurrency and Consistency*, chapter 13, page 556. Oracle Database Documentation. Oracle Corporation, online edition, 2008. Part No.: B28318-05.

[33] D. V. Tassel. Assignment. `http://hhh.gavilan.edu/dvantassel/history/assignment.html`, 2008.

[34] J. Zhou, J.-C. Freytag, P.-A. Larson, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In C. Y. Chan, B. C. Ooi, and A. Zhou, editors, *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, volume 1, pages 533–544, New York, 2007. ACM. 1247540.