An Efficient Bill-Of-Materials Algorithm

Ahmad Khalaila, Frank Eliassen
Department of Computer Science
The University of Tromsø
9037 Breivika, Norway
{ahmad,frank}cs.uit.no

April, 1997

Abstract

A large class of linear recursive queries compute the bill-of-materials of database relations.

This paper presents a novel algorithm that computes the bill-of-materials of its argument's (database) relation. The algorithm uses a special join operation that accumulates the cost of composite parts, without constructing the transitive closure of the argument relation, thus saving time and space.

We argue that this algorithm outperforms existent algorithms in the order of the diameter of the graph represented in the argument relation. This is made possible by exploiting knowledge of the level each tuple of the argument relation belongs to.

Moreover, this algorithm in contrast to transitive closure based processing, produces data at a very early stage of the processing which renders it suitable for pipelined distributed data processing.

1 Introduction

Given a transitive closure operator denoted α that does not eliminate redundant paths, and the relations defined by the following relational schema:

$$Uses : \mathsf{Relation}[(part : oid, subpart : oid, level : integer)];$$
$$Base : \mathsf{Relation}[(part : oid, cost : real)];$$

where $Uses$ is transitively defined and has a tuple for each $(part, subpart)$ relationship. A composite part may be involved in many such tuples. The $Base$ relation has a tuple for each base part (i.e. a part which is not composed of any other parts).

To compute the Bill-Of-Materials (BOM) in a system that provides a transitive closure primitive, one normally submits a query that is equivalent to the following α-algebraic [Agra87]. expression:

$$\prod_{Uses.Part,C} (GroupBy_{Uses.Part,C=sum(Cost)}(Base \bowtie_{Base.Part=Subpart} (\alpha(Uses))))$$

An execution strategy for the above expression that is based on evaluating each operation in the (above) strict nested order incurs very high execution cost. This high cost is due to the intermediate construction of the transitive closure of $Uses$.

We argue that any execution strategy for BOM algorithms that constructs the transitive closure of its argument is a bad strategy, in particular when that closure is much larger than the given relation. Moreover, in pipelined processing systems one is in need of algorithms that produces data as soon as possible. The strict nested-order evaluation of the above query does not produce any data until after the evaluation of the transitive closure of the argument relation.

In [KhEB96] we presented a BOM algorithm that avoid the evaluation of the transitive closure operator, and produces data at a very early stage (compared to the transitive closure processing). The algorithm combines some of the operations mentioned above into one specialized join operation, called $CJOIN$. The $CJOIN$ operation does not use any knowledge about the level of the tuples in $Uses$.

In this paper we present a BOM algorithm (called $OBOM$) which is very similar to the one in [KhEB96], except that the $CJOIN$ operation of the new algorithm implicitly exploits level knowledge.

The $OBOM$algorithm is superior to the algorithm of [KhEB96] in the order of the diameter of the graph represented in the $Uses$ relation.

The implementation of the $OBOM$ algorithm is based on the $Uses$ tuples being grouped on *part*, and then the groups being sorted on *level*. Using such an order $OBOM$ computes BOM in only one call to $CJOIN$, as we shall show.

1.1 Related Work

A transitive closure operator for database queries was first proposed by Zloof in [Zloo75]. Since then it has been shown that linear recursive queries can be expressed by such an operator [JaAN87, ChHa82], and an extension of relational algebra that includes a transitive closure operator called α-algebra has been proposed in [Agra87].

Furthermore, Agrawal [Agra87] (as well as many others) proposed that specialized algorithms that exploit the knowledge of the physical database can be built into the database system to efficiently implement the transitive closure operator and some frequent applications of it.

Bill-of-materials (and similar) computations constitute a large class of linearly recursive database computations that occur frequently in database systems environments containing transitive relations. When such queries are applied to very large relations, their efficient processing become vital (e.g. for users that are highly dependent on them). Although all such computations can be expressed using the transitive closure operator (as has been illustrated above), evaluating the transitive closure is not necessary for the evaluation of such computations. Since such an evaluation often incurs a very high cost in terms of time and space, we would like to avoid it. This is very similar to avoiding the evaluation of the cross-product when join is being evaluated [SmCh75, Ullm88b]. Moreover, in pipelined processing system one tends to avoid processing algorithms and strategies that produces data very late. That is due to the tremendous amount if waiting such algorithms impose on the operators that consume their output.

Many efficient transitive closure algorithms have been developed for different computing environments [Tarj81, AgJa87, Lu87, BiSt88, IaRa88, AgJa88, VaKh88a, VaKh88b, AgDJ90, ChDe90, HoAC90, Jian90, Jako91, DaJa92]. However for large data volumes, where the graph representation of these data is very complex, generating the transitive closure for such data may be very costly. Whenever generating such a closure can be avoided it should be. In [KhEB96], we proposed closure-based BOM algorithms that avoid generating the transitive closure of the argument relation. This results in better performance, both in terms of time and space.

Combining the execution of many operations into one has been first proposed by Smith et al. [SmCh75], and since then has been adopted by nearly everyone working with query processing and optimization [JaKo84, Ullm88b]. The combined join ($CJOIN$) algorithm is the core of our BOM algorithms since it combines the accumulation of (intermediate partial) cost for composite parts using the cost of their subparts, with the binary matching normally applied in join operations, to avoid the intermediate construction of the transitive closure of the input relation.

In section 2 we present the IBOM algorithm from [KhEB96], then in section 3 the $OBOM$ algorithm is presented and analyzed.

Finally, in section 4 we present the experimental results of the two algorithms, and analyze their results.

2 The Iterative BOM Algorithm

To compute the bill-of-materials for all the composite parts present in the *Uses* relation, it is not necessary to perform the transitive closure operation present in the BOM expression above, since we are not interesting in the all-pairs transitive closure of the graph represented by the *Uses* relation.

Additionally, many of the operations involved in the above query, can be done in a combined join operation (called *CJOIN*). The operation tries to match the subpart attribute of each tuple in the *Uses* relation with the part attribute of each tuple in the *Base* relation. If a match occurs it partially performs the *sum* operation by accumulating the cost of a *Uses* composite part that have a subpart that match a base tuple. The cost for each part is accumulated in the cost attribute of the corresponding tuple of the temporary relation *Accum*, which states the identity and cost accumulated so far for each (composite) part. The relational schema of *Accum* is $\mathsf{Relation}[(part : oid, cost : real)]$;

When analyzing the composition relationship we found that some parts are not composed (i.e. they are atoms or base parts), some parts are composed only of base parts (we will call them 1^{st} level parts), some parts are composed only of base (i.e. 0-level) and 1^{st} parts (we call them 2^{nd} level parts), some parts are composed only of 0-level, 1^{st} level, and 2^{nd} level parts (we call them 3^{rd} level parts), and in general i^{th} level parts are composed only of parts from the levels below, i.e. 0-level, 1^{st}level, 2^{nd}level, $\ldots$, and $i-1$ level. Notice that the sets of parts from the different levels are disjoint.

Based on the above observation, the iterative BOM algorithm (*IBOM*) starts by computing the total cost for 1^{st} level parts, then the total cost for 2^{nd} level parts, and so on. In general, computing the total cost for parts from the i^{th} level, will be completed only after the total cost for all parts from all the levels below (i.e. $i-1, i-2, \ldots, 1$) have been computed. Therefore, a run of the iterative BOM algorithm consists of the subsequent phases $1, 2, \ldots, D$, where D denotes the diameter of the directed acyclic graph (DAG) as represented by *Uses*. In each phase the total costs for the parts from the corresponding composite level are computed. That is, in phase i the total costs for all the parts from level i are computed, and phase i (for $i > 1$) is preceded by phase $i-1$ and is followed by phase $i+1$ (for $i < D$). Such a BOM algorithm terminates after the D^{th} phase.

2.1 Implementation of $CJOIN$

In this section we develop the $CJOIN$ operation used in the $IBOM$ algorithm specified below. This operation takes as input three argument relations $Accum, Uses$, and $Base$, and delivers as output three relations $Accum, Uses$, and $NewBase$.

The tuples of $Uses$ are grouped by the *part* attribute, and those of $Base$ and $Accum$ are hashed on their *part* attributes.

The following four operations are needed to implement the $CJOIN$ operation. The signatures and informal semantics of these operations are given below:

- $match : oid, Base \rightarrow TupleOf(Base)$
  match takes a part identity as its first argument and the current $Base$ relation as it second argument, and returns the $Base$ tuple corresponding to its first argument.

- $accumulate : real, oid, Accum \rightarrow$
  the $Accum$ tuple corresponding to its second argument is looked up, and its cost attribute is incremented by the value of the first argument. If such a tuple does not exist, it is created and inserted into $Accum$ and its cost attribute is initialized to the value of the first argument.

- $mark\_del : TupleOf(Uses), Uses \rightarrow$
  this function puts a deletion mark on the $Uses$ tuple corresponding to its first argument. This operation shrinks the volume of $Uses$.

- $move\_2NewBase : oid, real, Accum, NewBase \rightarrow$
  this operation is called when the total cost for a composed part has been computed completely. It increments the cost attribute of the $Accum$ tuple corresponding to its first argument by its second argument, and moves it to $NewBase$. This is the operation that inserts the base tuples of the next phase (of the $IBOM$ algorithm) into $NewBase$.

The above operations are implemented on top of hash-based structures on $Base$ and $Accum$. Hash-based structures and algorithms have been designed mainly to speed up the join operation involved in the $IBOM$ algorithm [Brat84, Kits83].

The $CJOIN$ algorithm performs the join of $Uses$ and $Base$, reduces and reconstructs all its arguments relations, and partially computes the aggregate function sum, all in one run through the tuples of $Uses$, $Base$, and $Accum$.

The notations we use to specify our algorithms are self-explanatory. However, the following elaborations may be helpful:

- $Tupleof(\mathsf{Relation}[(T_1, \ldots, T_N)]$ is an instance of $T_1 \times \ldots \times T_N$,
- All types has an element denoted $\perp$, that stands for "undefined value",
- Any text following "–" in a line is a comment, and
- A $group \in Uses$ stands for the sequence of tuples having identical part identities.

Algorithm 2.1 The combined join algorithm: $CJOIN$

```
$CJOIN(Accum, Uses, Base) \equiv$
    VAR:
        $u : TupleOf(Uses)$; $b : TupleOf(Base)$;
        $u2Base : bool$; – false, if some tuples in a group are not deleted
        $acost : real$; – the cost accumulated so far, for current group
    Program:
            For $group \in Uses$ Do– for each group in Uses
                $u2Base \leftarrow true$;
                For $u \in group$ Do– for each tuple in current group
                    $b \leftarrow match(u.subpart, Base)$;
                    If $b \neq \perp$ – is there a match ?
                        $acost \leftarrow acost + b.cost$;
                        $mark\_del(u, Uses)$; – delete the tuple
                    Else
                        $u2Base \leftarrow false$;
                If $u2Base$
                    $move\_2NewBase(u.part, acost, Accum, NewBase)$;
                Else If $acost \neq 0$
                    $accumulate(acost, u.part, Accum)$;
                $acost \leftarrow 0$
            Return$(Accum, Uses, NewBase)$;
```

2.2 Notations and assumptions

In the sequel we will use the following notations and assumptions:

- $|Uses| = N$, denotes the number of tuples of the $Uses$ relation;
- $|Uses^i| = N^i$, denotes the number of (remaining) tuples in $Uses$ at the end of the i^{th} phase;
- I is the number of distinct part identities that occur in the part attribute of $Uses$; i.e. the number of groups in $GroupBy_{part}(Uses)$;
- $|Base| = M$, denotes the number of tuples initially in $Base$;
- $|Base^i| = M^i$, denotes the number of tuples in $NewBase$ at the end of the i^{th} phase;
- The auxiliary operations *match*, *move_2NewBase*, and *accumulate* have a constant cost, denoted by C_0, while the others have a negligible cost. C_0 actually denotes the cost of accessing a tuple in *Base* or *Accum*;
- C_1 denotes the cost of accessing a $Uses$ tuple;

A simplifying assumption that otherwise has no major implication is the following:

Assumption 2.1 *(Uniform CJOIN behavior) The complexity of CJOIN behavior at the different D phases is uniform. That is, the same number of tuples are added to new Base, and Accum and the same number of tuples are deleted from Uses, at each phase.*

2.3 Implementation of the iterative BOM algorithm

The iterative BOM algorithm can be seen as a loop of joins between the *Base* and the *Uses* relations, each of which corresponds to a phase, as defined above. In each iteration the contents of the two relations will be changed, as explained in the sequel. Initially, the base parts will be those in *Base*, and *Uses* will have all the tuples representing the $(part, subpart)$ relation.

In the first iteration the total cost for all parts from 1^{st} level will be computed, the cost for all other parts that have some base subparts will be accumulated in *Accum*, every tuple in *Uses* that has a base subpart will be (marked) deleted, and the 1^{st} level parts together with their total costs comprise the new *Base* (denoted $Base^1$) of the next phase.

In the second iteration, the total cost for all parts from 2^{nd} level will be computed as above, and in general, in the i^{th} iteration the total cost of all parts from the i^{th} level will be computed, the cost of all other (i.e. higher levels) parts that have some base part components will be accumulated in $Accum$, every tuple in $Uses$ that has a $Base^{i-1}$ subpart will be (marked) deleted, and the i^{th} level parts together with their cost comprise the new $Base$ of the next phase (denoted $Base^i$).

The $IBOM$ algorithm depicted below constructs in each iteration (i) a new logically separated relation (fragment) to contain the new base tuples, and is called $Base^i$. That is, the base fragment $Base^i$ is constructed at the i^{th} iteration and corresponds to the $Base$ relation of iteration $i+1$. $Base^i$ contains a tuple for each of the i^{th} level part which has a part attribute corresponding to that i^{th} level part and a cost attribute whose value is the total cost of that part. $Base^0$ corresponds to the initial $Base$ relation which is used in the first iteration.

The temporary relation $Accum$ will at the end of each iteration i contain the cost for each j^{th} level ($j > i$) part which have some subpart from the levels below i. Within the i^{th} iteration, when the total cost for a level i part is computed, the $Accum$ tuple corresponding to that part, is moved from $Accum$ to $Base^i$.

Finally, the $Uses$ relation will at the end of each iteration i, have no tuple with a subpart from level i or any level below.

Algorithm 2.2 An Iterative BOM algorithm

```
$IBOM(Uses, Base^0) \equiv$
    VAR:
        $Accum, result$ : Relation$[(part\_id : oid, cost : real)]$;
        $i : integer$; – a phase counter
    Program:
            $i \leftarrow 1$;
            $result \leftarrow Base^0$
            While($Uses^i \neq \emptyset$) Do
                ($Accum^i, Uses^i, Base^i) \leftarrow CJOIN(Accum^{i-1}, Uses^{i-1}, Base^{i-1})$;
                $result \leftarrow result \cup Base^i$;
                $i \leftarrow i + 1$;
            Return($result$);
```

2.4 The Cost Formula of *IBOM*

The cost formula for $CJOIN$ is defined as follows:

$$CF_{CJOIN} = N \times C_1 + N \times C_0 + (I - I/D) \times C_0 + (I/D) \times C_0$$

In the above formula, the first and second terms denote the cost of the hash-based join operation. That is, the cost of accessing the tuples of $Uses$ and $Base$.

The third term, $(I - I/D) \times C_0$, corresponds to the (worst case) cost of accessing the *Accum* tuples in order to accumulate the cost of their corresponding parts. The fourth term $(I/D) \times C_0$ corresponds to the cost of restructuring *Accum* and $NewBase$.

Notice that the number of tuples in *Accum* will never exceed the number of groups in $Uses$ (i.e. I) minus the number of groups for which a total cost is emerging (i.e. I/D). Moreover, the number of tuples in $NewBase$ will never exceed I, in average it will be I/D.

Since $N > I$ is always true, the above formula is rewritten to:

$$\begin{aligned} CF_{CJOIN} &\leq N \times (C_1 + C_0) + I \times C_0 \\ &\leq N(C_1 + 2C_0) \end{aligned} \tag{1}$$

The cost formula for the iterative BOM algorithm can be expressed by using the cost formula previously developed for $CJOIN$, as follows:

$$\begin{aligned} CF_{IBOM} &\leq \Sigma_{i=1}^{D} N_i (C_1 + 2C_0) \\ &\leq (C_1 + 2C_0) \Sigma_{i=1}^{D} (N_i) \end{aligned} \tag{2}$$

The above formula is derived simply from the fact that in a run of $IBOM$ there is an $CJOIN$ call (whose cost is defined by equation 1) for each of the D levels in the DAG represented by $Uses$.

The term $C_1 + 2C_0$ in CF_{IBOM} involves only constants and therefore cannot be reduced further. However, using assumption 2.1, we may set $N_i = N - (i-1)N/D$. The term $\Sigma_{i=1}^{D}(N_i)$ can then be reduced as follows:

$$\Sigma_{i=1}^{D}(N_i) = N(D+1)/2 \tag{3}$$

Finally, by substituting equation 3 into equation 2 (i.e. $N(D+1)/2$ for $\Sigma_{i=1}^{D}(N_i)$) we get:

$$CF_{IBOM} \leq (C_1 + 2C_0)(D+1)N/2 \tag{4}$$

3 The *OBOM* algorithm

This section presents our new and very efficient algorithm (called $OBOM$) which is developed by implicitly using the knowledge of the level to which a tuple belongs.

The database schema consists of the following:

$$Uses : \mathsf{Relation}[(part : oid, subpart : oid, level : integer)];$$
$$Base : \mathsf{Relation}[(part : oid, cost : real)];$$

The algorithm assumes that the tuples of $Uses$ are grouped using the part attribute, and then the groups are sorted based on the *level* attribute. This ordering results in having all the groups of parts belonging to the first level to be located at the start of $Uses$, followed by all the groups of parts belonging to the second level, and so on until the end of $Uses$ where all the groups of parts belonging to level D are located. That is, the group of tuples determining the cost of each part from a level j are grouped together and occur (in $Uses$) before any group from any level $k > j$, and after any group from any level (i¿j).

Moreover, $Base$ is hash-structured and contains initially a tuple for each part from level 0.

The $OBOM$ algorithm is very similar to $CJOIN$ but much simpler as a result of the knowledge it implicitly possesses about the ordering of tuples in $Uses$. The algorithm uses two routines, *match* which has the same functionality as in $CJOIN$, and *hash_insert* which inserts a new base tuple into $Base$.

Algorithm 3.1 A very efficient BOM algorithm: *OBOM*

```
$OBOM(Uses, Base) \equiv$
    VAR:
        $u : TupleOf(Uses)$; $b : TupleOf(Base)$;
        $acost : real$;
    Program:
            For $group \in Uses$ Do– for each group in Uses
                For $u \in group$ Do– for each tuple in current group
                    $b \leftarrow match(u.subpart, Base)$;
                    $acost \leftarrow acost + b.cost$;
                $hash\_insert((u.part, acost), Base)$;
                $acost \leftarrow 0$;
            Return($Base$);
```

Algorithm 3 starts by computing the total cost of parts from the first level, and since all their subparts are (from level 0 and therefore already) in $Base$, *match* will never fail to match a corresponding base tuple. After the cost of a part is computed it is inserted into $Base$. Consequently, when the costs of all parts from the first level are computed, they are stored in $Base$, hence computing the costs of parts from the second level can start, and so on. In general, when $OBOM$ starts computing the costs of parts from level j, $Base$ already contains the total costs of all parts from all levels $i < j$. When the total costs of all parts from level D are computed, the algorithm reaches the end of $Uses$and terminates, and $Base$ contains the costs of all parts.

3.1 Complexity of $OBOM$

This algorithm accesses each tuple in $Uses$ only once, thus it is optimal with regard to its access to $Uses$. Because any BOM (or any transitive closure) algorithm will have to access the tuples of each group in order to compute the cost of their corresponding part. On the other hand, the algorithm accesses each tuple of $Base$ (not only the initial $Base$) a number of time equivalent to its frequency as a subpart in $Uses$. But that is also the minimum number of accesses needed to compute the cost of all parts. Since, a cost of a part is determined by the cost of its subparts, there is a need to access $Base$for each subpart in order to compute the total cost.

One way to optimize the accesses to $Base$, is to cluster all the $Uses$

tuples having the same subpart, but then we may have destroyed the access structure imposed on $Uses$ and that made this algorithm possible. Moreover, a need for temporary accumulation will arise, as in $IBOM$.

In $IBOM$ each invocation i of $Cjoin$attempt to match each $Uses$ tuple that is not marked deleted with a base tuple, to extract the cost of the subpart of that tuple of $Uses$. Such a match will fail for all $Uses$ tuples that have a subpart that is not currently in $Base$(i.e. a subpart that belongs to a level $j >= i$).

The complexity of the $OBOM$algorithm is defined as follows:

$$(N \times C_1) + N \times C_0 = N \times (C_1 + C_0)$$

which is superior to $IBOM$ in the order D.

For many environments in which BOM computations are critical and vital to their operation, it seems to us worth to maintain the knowledge of the level of parts in $Uses$. Maintaining such knowledge can be done very efficiently and in an incremental manner, hence enabling the application of this new algorithm.

4 Experimental results and their analysis

This section presents the results of a lab experiment which tries to infer a correspondence between the results of the theoretical analysis and empirical facts. In other words, we looked for empirical facts to refute the result of the theoretical analysis. That is, if the performance of $OBOM$ is actually superior to $IBOM$ in the order of D.

4.1 The lab environment

We implemented the $IBOM$ and $OBOM$ algorithms in C/Unix. We ran the $IBOM$ and $OBOM$ programs on a HP-UX 9000/780 (C-160) machine, having 128 Mb memory and 4 Gb disk space. It should be noted that the compilation of the programs did no optimization for this architecture. That is, the performance results (i.e. response time for the various runs) should not be perceived as being the best results obtainable on this architecture. This is acceptable since we are conducting a comparative study of two algorithms, rather than trying to find the best time achievable by these algorithms on a specific architecture.

The $IBOM$ or $OBOM$ program ran on the system alone, i.e. there were no concurrent user processes on the system.

4.2 The construction of test data

A program called mk-graph constructs the data for the experiment. This program takes 4 arguments; the number of parts in the graph (denoted N), the number of levels in the graph (denoted L), the minimal number of subparts in each composite part (denoted C), and the minimal number of parts a part is a subpart of (denoted P).

The program constructs both $Uses$ and $Base$. It constructs $Uses$ by virtually building a directed acyclic graph (DAG) having:

- D levels,
- each part at any level (except level D) is engaged as a subpart in at least P tuples in $Uses$, and
- each part at any level (except level 0) is engaged as a composite part in at least C tuples in $Uses$.

The program assigns N/D parts to each level as follows. It assigns the parts $1..(N/D)$ to level 0, then the parts $((N/D)+1)..(2N/D)$ to level 1, and so on until finally the parts $(D-1)N/D..N$ are assigned to level $D-1$.

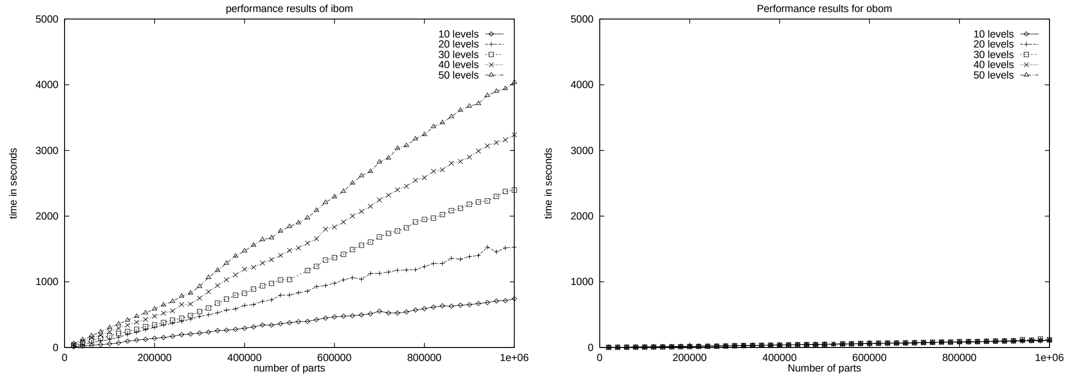
The construction of $Base$ is much simpler. mk-graph constructs a tuple for each part of level 0, and attaches to it a cost value which is chosen pseudo-randomly.

In this way the program can control the volume of data in the graph (i.e. the number of parts) and its complexity (i.e. the number of $Uses$ tuples a part is engaged in as a composite part or as a subpart).

4.3 The tests and their analysis

We want to test the hypothesis that the $OBOM$ algorithm is superior to the $IBOM$ algorithm in the order of D. Since we are addressing large database processing, we also want to test the impact of large data volumes and large number of levels on the performance of these algorithms.

In our experiment, both N and D varies, while P and C remain unchanged having the value 10 throughout the whole experiment. The size of a $Uses$ tuple also remains unchanged. Thus, neither the impact of graph complexity nor that of the tuple size is considered directly. The reason for this being that our analysis shows that these factors merely increase the size of the data. Testing the impact of the size of data on the performance of the algorithms should therefore prove sufficient. Figure 1(a) depicts the performance results of $IBOM$. The figure depicts the response time for a



(a) IBOM

(b) OBOM

Figure 1: IBOM and OBOM performance

series of runs of $IBOM$ that are performed for different number of levels and different number of parts.

From figure 1(a) we conclude first, that the response time of $IBOM$ increases in a linear proportion to the size of data. Second, that the number of levels in the graph has a major impact on the performance of $IBOM$. There is a linear increase in response time proportional to the number of levels.

Figure 1(b) depicts the performance results of $OBOM$. The figure depicts the response time for a collection of runs of $OBOM$, performed using different values of N and L.

Based on the data shown in figure 1(b) we conclude first, the performance of $OBOM$ is completely independent of the number of levels, and second, a very weak linear increase in response time is observed as the volume of data increases.

4.4 Conclusion

By comparing the performance results of $IBOM$ to those of $OBOM$, we find that $OBOM$ is superior to $IBOM$ in the order of D. Thus, our theoretical hypothesis (i.e. the result of the complexity analysis) corresponds to the empirical facts.

However, the correspondence is only inferable as long as the whole result of $OBOM$ can be contained in main-memory. Recall that $IBOM$ needs to store in memory only a fragment of $Base$, (i.e. the fragment that have been produced in the previous call to $Cjoin$) while $OBOM$ stores the entire $Base$.

5 Acknowledgment

We wish to thank Åge Kvalnes, Espen Skoglund and Kjetil Jacobsen for their efforts in writing a Perl script that generated the result data, Ken Hirsch for his interest in BOM computation which inspired the development of this new and efficient algorithm, and Gaute Nessan for very interesting discussions on computer systems in general and Unix in particular. He also helped find some C bugs.

References

[Agra87] Agrawal, R., "Alpha: An extension of Relational Algebra to Express a class of Recursive Queries," Proc. 3rd Int'l Conf. on Data Engineering, February 1987.

[AgJa87] Agrawal, R., Jagadish, H.V.,"Direct Algorithms for Computing the Transitive Closure of database relations," in Proc. 13th Int'l Conf. on VLDB, 1987.

[AgJa88] Agrawal, R., Jagadish, H.V.,"Multiprocessor Transitive Closure Algorithms," in Proc. Int'l Symp. on Databases in Parallel and Distributed Systems, Austin, Texas, Dec. 1988, pp. 56-67.

[AgDJ90] Agrawal, R., Dar, S., Jagadish, H.V., "Direct Transitive Closure Algorithms: Design and Performance Evaluation," in ACM Trans. on Database Systems, 15(3), Sept. 1990.

[Banc85] Bancilhon, F., "Naive Evaluation of Recursively Defined Relations," TR. DB-004-85, MCC, Austin, Texas, 1985.

[BiSt88] Biskup, Stiefeling, " Transitive Closure Algorithms for Very Large Databases," TR, Hochschule Hildesheim,1988.

[Brat84] Bratbergsengen, K., "Hashing Methods and Relational Algebra Operations," Int'l Conf. on VLDB, Singapore, Aug. 1984.

[ChDe90] Cheiney, J., De Maindreville, C., “ A Parallel Strategy for the Transitive Closure Using Double Hash-based Clustering,” in Proc. Int'l Conf. on VLDB, aug., 1990.

[ChHa82] Chandra, A.K., Harel, D., “Horn clauses and the fixpoint query hierarchy,” Proc. 1st Symp. Principles of Database Systems, 1982, pp. 158-163.

[Codd70] Codd, E.F., “A relational model of data for large shared data banks,” CACM, vol. 13, June 1970, pp. 377-387.

[Codd72] Codd, E.F., “Relational completeness of database sublanguages,” DataBase Systems, R. Rustin, Ed. Englewood Cliffs, NJ:Prentice-Hall, 1972, pp. 65-98.

[DaJa92] Dar, S., Jagadish, H.V., “A Spanning tree Transitive Closure Algorithm,” in Proc. 8th Int'l IEEE Conf. on Data Engineering, 1992.

[Graf93] Graefe, G., “Query Evaluation Techniques for Large Databases” ACM Computing Surveys 25 (2), June 1993.

[Gutt84] Guttman, A., “New Features for Relational Database Systems to Support CAD Applications,” Computer Science Dept., Univ. of California, Berkeley, June 1984, Ph.D. Dissertation.

[HoAC90] Houtsma, M., Apers, P., Ceri, S., “ Distributed Transitive Closure Computation: the Disconnection Set Approach,” in Proc. 16th Int'l Conf. on VLDB, Aug., 1990.

[IaRa88] Ioannidis, Y.E., Ramakrishnan, R., “Efficient Transitive Closure Algorithms,” in Proc. 14th Int'l Conf. on VLDB, 1988.

[Ioan86] Ioannidis, Y.E., “On the Computation of the Transitive Closure of Relational Operators,” Proc. 12th Int'l. Conf. on Very Large Date Bases, August 1986, pp. 403-411.

[JaAN87] Jagadish, H. V., Agrawal, R., Ness, L., ”A Study of Transitive Closure as a Recursion Mechanism,” Proc. ACM-SIGMOD 1987 Int'l Conf. on Management of Data, May 1987.

[JaKo84] Jarke, M., Koch, J., “Query optimization in database systems,” ACM Computing Surveys, 16(2), pp 111-152, June 1984.

[Jako91] Jakobsson, H., “Mixed-approach Algorithms for Transitive Closure,” in Proc. of ACM Symp. on PODS, Denver, Co., May, 1991.

[Jian90] Jiang, B., “ A Suitable Algorithm for Computing Partial Transitive Closures in Databases,” in Proc. IEEE Conf., on Data Engineering, 1990.

[Khal96] Khalaila, A., “Partial Evaluation and Early Delivery: Adapting the Pipeline Processing Strategy to Asynchronous Networks,” PhD Dissertation, The University of Tromsoe, 1996.

[KhEB96] Khalaila, A., Eliassen, F., Beeri, C., “Efficient Bill-Of-Materials Algorithms,” Technical Report 96-25, The University of Tromsø, Sept. 1996.

[Kits83] Kitsuregawa, M., et al., “Applications of Hash to Data Base Machine and Its Architecture,” in New Generation Computing, vol. 1, 1983.

[Lu87] Lu, H., “ New Strategies for Computing the Transitive Closure of Database Relations,” in Proc. 13th Int’l Conf. on VLDB, 1987.

[SACLP79] Selinger, P. G., Astrahan, M.M., Chamberlin, D.D., Lorie, R. A., Price, T.G., “Access Path Selection in a Relational Database Management System,” ACM-SIGMOD 1979.

[SmCh75] Smith, J.M., Chang, P.Y., “Optimizing the Performance of a Relational Algebra Database Interface” CACM 18(10), October 1975.

[Tarj81] Tarjan, “Fast Algorithms for Solving Path Problems,” Journal of the ACM, 28(3), 1981.

[Ullm88a] Ullman, J.D., Principles of Database and Knowledge Base Systems, Vol. I, Computer Science Press, Rockville, Md., 1988.

[Ullm88b] Ullman, J.D., Principles of Database and Knowledge Base Systems, Vol. II, Computer Science Press, Rockville, Md., 1988.

[VaKh88a] Valduriez, P., Khoshafian, S., "Transitive Closure of Transitively Closed Relations," 2nd Int'l Conf. on Expert Database Systems, L. Kerschberg (ed.), Menlo Park, Calif., Benjamin-Cummings, 1988, pp. 377-400.

[VaKh88b] Valduriez, P., Khoshafian, S., "Parallel Evaluation of the Transitive Closure of a Database Relation," Int'l Journal of Parallel Programming, Vol. 17, No. 1, Feb., 1988.

[Zloo75] Zloof, M.M., "Query-By-Example: Operations on the Transitive Closure," RC 5526, IBM, Yorktown Hts, New York, 1975.