Global Optimization and Parallelization of Integrity Constraint Checks

Uwe Herzog * *Jörg Schlösser*

Universität Karlsruhe
Fakultät für Informatik
D–76128 Karlsruhe, Germany
herzog|joerg@ira.uka.de

Abstract

In this paper we consider the problem of efficiently checking integrity constraints. We concentrate on a combination of two optimization techniques: global optimization and parallelization. The primary objective of global optimization is to produce a global execution plan for a group of constraints where common subexpressions can be shared. Since parallelism is a promising strategy for improving efficiency in database systems we apply parallelism to integrity constraint checking. In a first step we compile the declarative constraints into appropriate constraint checks using an extended relational algebra. Then, the algebraic expressions are forwarded to our optimizer. Because many constraints have to be checked at the same time, considerable savings are possible in a second step by identifying common subexpressions in order to evaluate them only once. In a third step we generate parallel constraint execution plans, that are to be executed by a parallel constraint engine which supports a dataflow–driven, set–oriented constraint execution. Horizontal and vertical parallelism is supported through data distribution and pipelining between extended relational algebra operators. We demonstrate the savings in processing costs that can be achieved by factorization and parallelization of integrity constraint checks by measurements performed with a parallel main–memory database system.

*This work was supported in part by the Graduiertenkolleg "Controllability of Complex Systems" (DFG Vo 287/5-2).

1 Introduction

The reliability of information systems is a major requirement for today's society and enterprises. In this context, the correctness or semantic integrity of databases is one of the main issues. One way to capture the semantics of application domains is to specify integrity constraints in a declarative manner. Ideally, these integrity constraints have to be proven automatically by the underlying database management system. It is still an open problem how to do this efficiently. To contribute towards a solution of this problem is a chief focus of our research. Today the prime obstacle to applying proof procedures is their high computational costs. Checking integrity constraints may be inefficient for several reasons, e.g. the inherent complexity of constraints, the huge data volume, the high update frequency of data and the ensuing high check frequency of constraints, and the large set of constraints. The basic idea underlying our Comet project (**C**onstraint **o**ptimization **met**hods) is to attack the problem of integrity constraint checking by interpreting constraint optimization as a special case of query optimization.

1.1 Project Description

Even though constraints may be viewed akin to database queries, there is a significant difference: queries are posed spontaneously whereas integrity constraints are usually determined at schema definition time and tend to remain fairly stable over time. This allows us to base our approach on the optimization of constraints at compile time and hence to minimize the amount of computation that has to be done at run time. Our basic concept of integrity checking is described in figure 1.

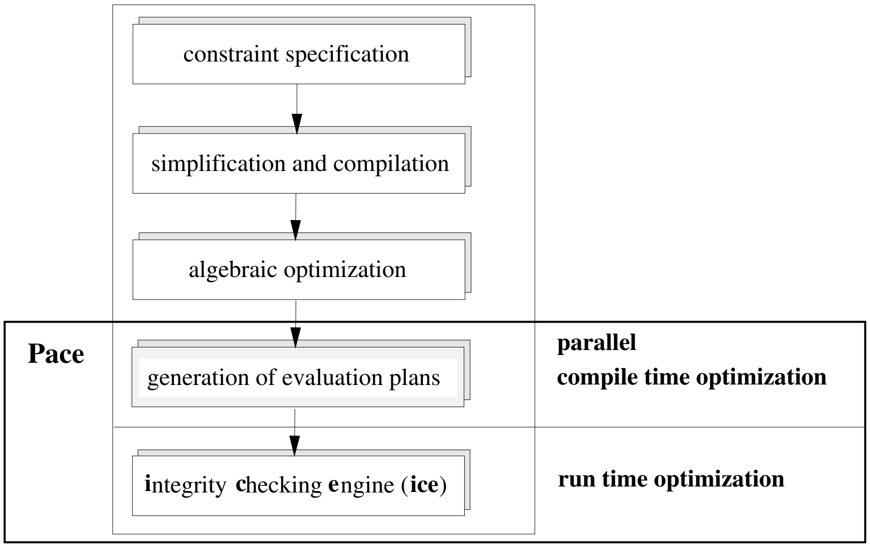

Figure 1: Basic concept of integrity constraint checking

Essentially, two phases can be distinguished. The first phase consists of all steps which are applied just once, namely at definition time of the integrity constraint. All steps of the second phase are applied as part of the execution of the integrity check. Therefore, the optimization steps used in our approach are divided in two phases. A main goal of the approach is to apply most of the optimization steps during the first phase.

We assume integrity constraints to be closed range–restricted first–order logic formulas. In constraints we allow base predicates (relations), derived predicates (rules) and computable predicates (e.g. comparison operations)[1]. Provided that the database is consistent at the beginning of a transaction, simplified integrity constraint checks [Nic82] are generated. In our approach, these are based on difference relations for inserted (deleted) tuples. Since these can be introduced at definition time, the optimizer need not to know the transactions which will be the cause of the subsequent checks. The integrity constraints are then compiled from logical formulas into expressions of an *extended relational algebra* [HM94]. To account for special patterns of integrity constraints we introduce new operators in the algebra that, at the same time, are particularly amenable to parallelization. Expressions in this extended relational algebra are optimized in nearly the same way as queries, although the cost model may differ to some extend from the customary ones for queries. The last step at compile time is the generation of evaluation plans. As a result we obtain one or more optimized parallel evaluation plans. At run–time one of them is selected and submitted to a system called the **I**ntegrity **C**hecking **E**ngine (ICE, see [HS95]), which we developed for the execution of parallel evaluation plans. The optimizer for generating parallel execution plans and the ICE are parts of the Pace project (**Pa**rallel **C**onstraint **E**ngine).

Problem Statement: Because a database update usually affects more than one integrity constraint, it is often the case that a group of constraints must be submitted together to the constraint execution engine for evaluation. The traditional single constraint optimization approach of evaluating these constraints one at a time is likely to be inefficient when the constraints share common relations. Global optimization can be used to identify common subexpressions among constraints and create a global constraint execution plan. In this plan common subexpressions of different constraints to be checked are evaluated only once.

The problem is closely related to the multiple–query optimization problem. Still, there are some differences: First, constraints are represented in our extended relational algebra which contains operators with two output sets. The second output set can be computed without additional work. Therefore, a group of constraints should be examined for different operators which can be computed together in order to reduce the total work. Because these are no common subexpressions in the traditional sense we called this process *factorization*. Second, in real applications many relationships exist between entities modelled in the database. Often, the mapping of this relationships into integrity constraints results in complex integrity constraint checks. Consequently the evaluation of these constraints is expensive. Third, the evaluation frequency of constraints can be very high since one transaction can trigger many constraint checks. Consequently, optimization is very important and little optimization gains can be worthwhile over the time, too. Fourth, in opposite to queries the definition of

[1] At the moment, we do not support aggregate functions in constraints.

the constraints is independent from the evaluation of the constraints. Therefore, expensive methods can be used during optimization.

1.2 Related Work

In this paper we consider the use of optimization techniques and parallelism to deal effectively with the requirement of high–performance constraint checking. The work in the field of integrity checking mainly considered the simplification of integrity constraints [Nic82]. Little attention has been paid to the execution of integrity constraints, even though this is known to raise problems [GA93].

Because we deal with constraints as queries, we have to integrate optimization and parallelization techniques. The problem of multiple–query optimization (or global optimization) has been examined in the past. In [SG90] the complexity of the multiple–query optimization problem is studied and it is shown that the problem is NP–hard. Sellis [Sel88] suggests to use heuristic algorithms in order to solve this problem. The input to the optimizer is a limited set of all equivalent execution plans for each single query but it is not obvious which of them are good candidates. Other papers examine the performance of heuristic algorithms to solve the multiple–query optimization problem [CLS93, SSN94].

For years now, attention has been paid to use parallelism to achieve high–performance in database systems [DG92]. Typically, queries are represented as operator trees. Optimization of such trees was an important issue in the work of several authors [HS91, SE93]. A second focus of research was the parallelization of the join operator [DG85]. By contrast, the field of parallel constraint checking has so far rarely been mentioned in research. The first work in the field of parallel constraint checking is [GFA92] within the PRISMA/DB project. However, only simple classes of integrity constraints are examined. In [MK94] simple constraints are considered, too, where parallelism is restricted to horizontal parallelism.

Contribution of our work: The main objective of this paper is to present an integrated solution for combining factorization and parallelization in an optimizer for efficient execution of integrity constraints. Since the search space for the optimal plan is usually very large, we propose a two–step–approach for reducing the optimization time.

1.3 Outline of Paper

The remainder of this paper is organized as follows. In section 2 we present our concept of the optimizer. In sections 3 and 4 our optimization techniques are discussed: the global optimization of integrity constraints through factorization, and the parallelization of the execution plans. In section 5 we demonstrate the impact of our optimization techniques. In section 6 we draw some conclusions and discuss future directions of our work.

2 Basic Concept of the Optimizer

The goal of the optimizer is to select an efficient execution plan for a group of constraints from a set of feasible plans. The optimization objective is minimizing the response time. The constraint execution plan specifies the physical operators (the methods used to implement the operations), the operands, their execution order and the degree of parallelism. Our optimization problem is expected to be very complex because three dimensions of optimization are introduced: single constraint optimization, optimization of a group of constraints, and parallel optimization. Each of these problems is known to be a hard search problem and, therefore, all approaches try to avoid the investigation of a complete search space [SSN94, SE93]. One way is to construct only a part of the whole search space. Another way is to use heuristic search strategies which do not consider the whole search space but deal only with selected points of the search space. A third way is to divide the optimization process into phases and starting each phase with an optimal result from the previous phase. A main challenge there is to guarantee that the combination of the local optima comes to the global optimum.

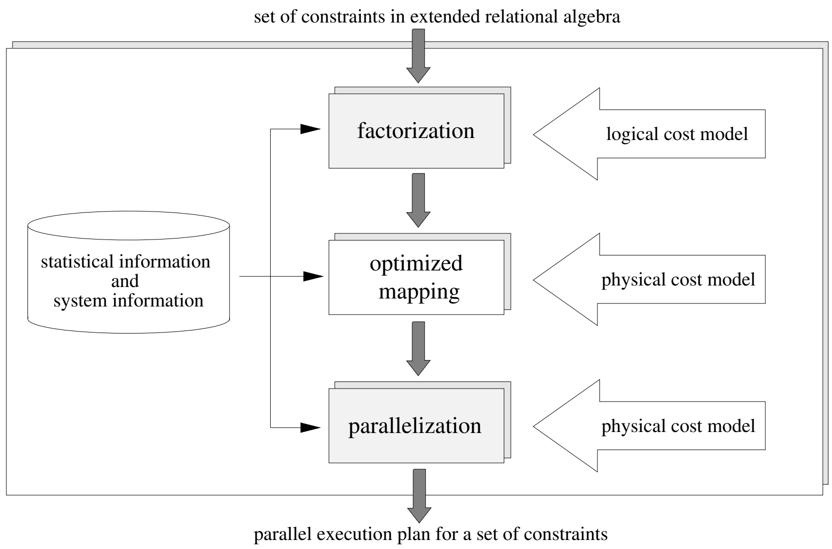

Figure 2: Optimizer Overview

Because of the three optimization dimensions, the number of feasible plans increases dramatically. Therefore, it is also necessary to restrict the global search space for the optimizer. We propose the division of the optimization process in three phases, analogous to strategies in other optimizers which split the optimization into a sequential and parallel optimization step [HS91].

The principal steps of the optimizer are the *factorization* (see section 3), the *optimized mapping* and the *parallelization* (see section 4). The input to the optimizer are groups of constraints as expressions in extended relational algebra. The algebraic expressions from the constraints in first–order logic are generated by our Comet constraint compiler [HM94].

The factorization and parallelization steps work with different internal representations of the group of constraints. The internal representation of the factorization step is a logical multi dataflow–graph. This representation is suitable for detecting common subexpressions in different constraint checks. The optimization goal of this phase is minimizing the total amount of work.

In order to connect the factorization step to the parallelization step we introduce an intermediate step called optimized mapping, which generates a physical dataflow graph from the logical dataflow graph. To do so we have to solve two types of problems. First, we must determine the operator implementation, e.g. the physical operator for each logical operator in the dataflow graph. Second, we must determine the order of operators within sequences of identical operators (e.g. join order), because this ordering is not entirely fixed by the results of the factorization step. The first step depends on the available set of physical operators of the integrity constraint engine (ICE [HS95]). The second step is an optimization step because of the mapping of logical operators with n operands to physical operators with 2 operands. Due to the associativity and commutativity of the join operator many possibilities exist for the mapping. Therefore, we use a heuristic which is similar to the well–known minimum–selectivity–heuristic. While the original heuristic tries to minimize the size of the temporal relations, we use a variation of this heuristic which tries to minimize the completion time of an operator. This follows directly from the optimization goal for parallel execution which is minimization of execution time as opposed to minimization of work in the case of sequential execution. Due to this heuristic, this approach differs from other approaches in the the field of query parallelization where the starting point for the parallelization is the "best" sequential plan. Here, we use several starting points because of the specific representation of the factorization result. The parallel cost model used is sketched in section 4.

In the parallelization step for each operator the degree of parallelism is selected. The optimization goal, as just mentioned, is to minimize the execution time of the constraint evaluation.

3 Global Optimization through Factorization

3.1 Factorization as Optimization Technique

The motivation for *factorization* is to exploit the benefits of sharing common tasks (or subexpressions) in the execution plans for a group of integrity constraints. Since there is a high possibility that constraints which are initiated by the same transaction share common relations and predicates, the traditional single constraint optimization approach is likely to lead to an inefficient execution. The factorization step of the optimizer generates an integrated execution plan for a group of constraints in which the common tasks of different constraints are evaluated only once. Due to the treatment of constraints as queries, our factorization is strongly related to a similar field in the context of queries which is often called *global optimization* in the literature. According to [PS88] global optimization has to deal with two problems:

1. *Discovering common subexpressions:* The objective of this subproblem is to identify related subexpressions of multiple queries which can be substituted by one expression. The result of this expression may be used in a shared manner by the multiple queries. Alsabbagh and Raghavan [AR94] are considering the following 4 relations between two subexpressions: Two subexpressions are related by *equivalence* (the results are identical), are related by *implication* (the result of one is a proper subset of the other), are related by *intersection* (the intersection of the results is not empty), or are not related at all. The discovery of common subexpressions is made more difficult due to hidden subexpressions. Potential common subexpressions are hidden by selections or different join orders. In order to discover such hidden common subexpressions, transformations of initial queries are necessary.
2. *Using common subexpressions (constructing a global access plan):* After discovering common subexpressions in different queries, the queries have to be merged into a unified query. This should be done only if the unified query has lower (estimated) execution time compared to the sum of execution times of the different queries. Often there exists more than one possibility for building a unified query because more than one common subexpression is discovered. In this case the unified query with the lowest execution time has to be selected.

Basically, the same problems apply to our factorization of constraint checks. However, there are some differences to global optimization, too:

- In contrast to queries the set of integrity constraints is known (long) before the execution of the constraints. Therefore, the factorization can take place directly after the definition of the constraints.
- A main problem of global query optimization is to determine the set of queries which should be considered in order to exploit common subexpressions. In the case of global optimization of constraints this set can be selected statically under consideration of common relations (see [HM94] for details).
- In order to achieve efficient execution, one may introduce special operators with two complementary output sets. Therefore, in addition to the four relations between subexpressions mentioned above, we consider subexpressions which are related by *complement* (the results of one subexpression can be calculated by using the complement of the other subexpression).

3.2 Internal Representation: The Logical Multi Dataflow Graph

For the factorization of constraints, a representation which abstracts from the ordering and the specific implementation of operators is needed. Our representation for this task is based on operator graphs, and is defined as follows:

Definition 3.1 (LMDFG) *Constraints are represented as a **L**ogical **M**ulti **D**ata **F**low **G**raph which is a directed, acyclic, connected graph. Internal nodes*[2] *are representing logical operators which are generalizations of the algebraic operators with regard to the number of operands and results they could use or produce, respectively. An edge from a node A to a node B has the meaning, that node B*[3] *uses the result of node A. Because nodes may have multiple results, edges are labeled with an identification for the specific result they refer to. Each LMDFG contains one specific node named **Execute** which has no outgoing edges. This node is the sink of the graph and represents the results of the various constraints. The nodes which have no incoming edges (the sources of the graph) are standing for the base relations.*

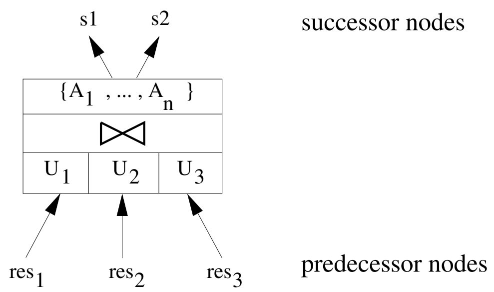

Figure 3: Join–node of a LMDFG with 3 operands

Furthermore, for each result of a logical operator an additional *projection set* might be included in the node. For example, the part of a LMDFG in figure 3 represents the following equivalent sequences of XRA–operators:

$$\begin{array}{ll}
\pi_{A_1,\ldots,A_n}(((res_1)_{U_1} \bowtie (res_2)_{U_2}) \bowtie (res_3)_{U_3}), & \pi_{A_1,\ldots,A_n}((res_1)_{U_1} \bowtie ((res_2)_{U_2} \bowtie (res_3)_{U_3})), \\
\pi_{A_1,\ldots,A_n}(((res_1)_{U_1} \bowtie (res_3)_{U_3}) \bowtie (res_2)_{U_2}), & \pi_{A_1,\ldots,A_n}((res_1)_{U_1} \bowtie ((res_3)_{U_3} \bowtie (res_2)_{U_2})), \\
\pi_{A_1,\ldots,A_n}(((res_2)_{U_2} \bowtie (res_1)_{U_1}) \bowtie (res_3)_{U_3}), & \pi_{A_1,\ldots,A_n}((res_2)_{U_2} \bowtie ((res_1)_{U_1} \bowtie (res_3)_{U_3})), \\
 & \vdots
\end{array}$$

The $U_1, \ldots, U_3$, which may be specified in a node for each of its operands, are *renaming lists* for the attribute descriptors of the relations. They are needed, because our LMDFGs contain algebraic operators like the natural join whose semantic depends on the attribute names of the operands. Furthermore, the use of renaming lists preserve for each node in a LMDFG its specific view of the operands, especially when sharing some of its operands with nodes from different constraints. This makes the integration of multiple constraints in one LMDFG easier.

[2]These are nodes which have both, incoming and outgoing edges.

[3]Due to the limited space, we use the term "node" as a shortcut to express the logical operator that represents the node.

3.3 Principles of Factorization

Since subexpressions which are not related are of no further interest, we observe four principles for the factorization, corresponding to the 4 remaining relations, mentioned in section 3.1. These are factorizations, that use either *equivalence of nodes* for the equivalence relation, *implication of nodes* for the implication relation, *extraction of nodes* for the intersection relation, and *combination of nodes* for the complement relation.

Below we briefly describe each principle. As an example we use the join–operator in the figures, omitting details like projection sets, renaming lists, etc.. Similar figures could be drawn for other operators as well.

- *factorization using equivalence of nodes*

  We consider two nodes to be *equivalent*, if they have the same predecessors, use the same results of them, and the results are equal except for the attribute columns. Then, two such nodes can be replaced by one new node (figure 4), and the successor nodes share the results of the new node. Therefore, the number of nodes in the LMDFG is decreasing.

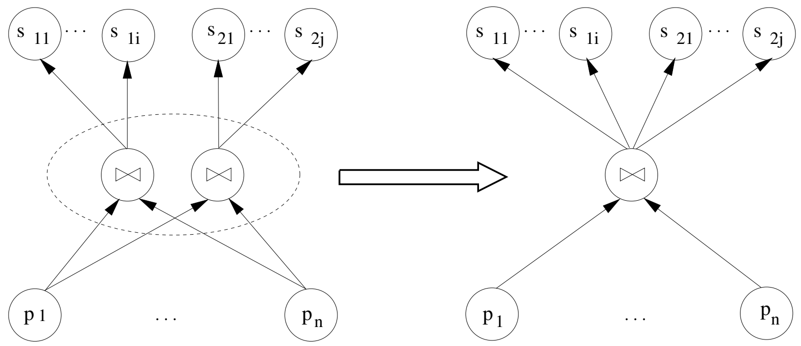

Figure 4: node equivalence

It is worth mentioning, that the projection sets included in the nodes need special attention during a manipulation of a LMDFG. For example, in figure 4 the two equivalent join nodes could have different projection sets. As a consequence, the projection set of the new join equals the union of the two original projection sets, after renaming some of the attributes to achieve a common namespace. This complies to shift parts of the original projections into successor nodes according to the laws involving projections in [Ull89]. Note, that sometimes we have to use the renaming lists to guarantee the correctness of the shifts. Because we only consider constraints without aggregates the shift of projections is always possible. Details are reported in [Sch95], and are omitted here and in the rest of this paper due to space restrictions.

- *factorization using implication of nodes*

  One node implies another, if the set of predecessors of the implying node is a subset of the predecessor set of the implied node, they share the results of their common predecessors, and the result of the implying node can be used by the implied node to reduce the workload to build its result. A LMDFG which includes such nodes can be altered by replacing the specific operands of the implied node with the result of the implying node (figure 5). The number of nodes remains the same.

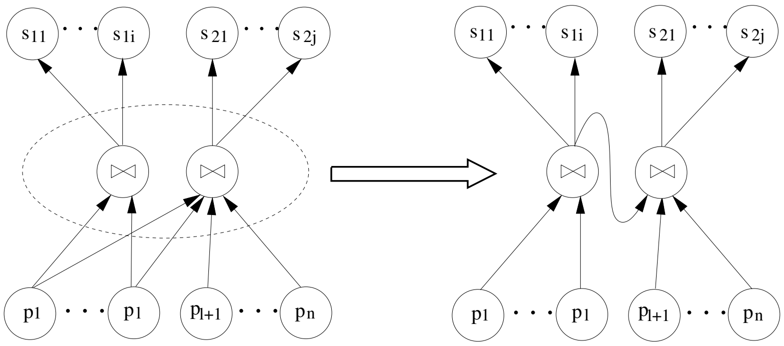

Figure 5: node implication

- *factorization using extraction of nodes*

  Sometimes it is possible to extract a new node from two nodes, and the result of the new node can be shared by the two nodes to produce their results. Figure 6 shows exemplary such a case for join–operators with at least 3 operands each. The new node possesses the common operands of the original nodes as operands.

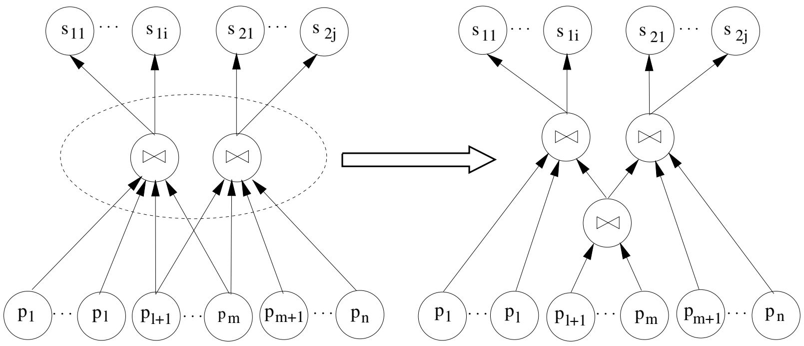

Figure 6: node extraction

The principle can be applied whenever the result sets of the original two nodes without attribute projection are not disjoint. Clearly, the number of nodes in the LMDFG increases after a manipulation using this principle.

- *factorization using combination of nodes*

  Due to special operators of the XRA [HM94] which produce complementary results, it is possible to replace two nodes which produce such complementary results by a new node which represents such a special operator. In figure 7 a situation is shown where a join ($\bowtie$) and anti–join operator ($\overline{\bowtie}$)[4] are combined into a node representing the operator $\bowtie_2$. The number of nodes in the LMDFG decreases after such a factorization.

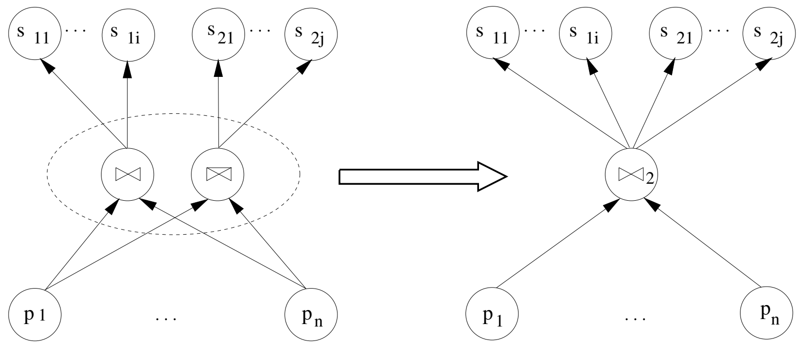

Figure 7: node combination

3.4 Principles of Manipulation

As mentioned in section 3.1, the detection of common subexpressions can be difficult due to an unfortunate order of operators. The factorization principles are able to detect common subexpression in a given LMDFG. But, maybe there are existing other possible common subexpressions in a equivalent LMDFG[5]. To detect such expressions it is necessary to transform the LMDFG into the equivalent one. For this purpose we use the following two *manipulation principles:* moving of selection nodes towards the execute node according to the algebraic laws, and merging of subsequent nodes of the same type, e.g., two connected join–nodes each with two incoming edges are merged in one join–node with three incoming edges. It is worth to mention that in contrast to the factorization principles the usage of the manipulation principles often lead to a LMDFG with a higher amount of work. This is appropriate when following factorization steps are leading to an LMDFG with lower amount of work due to detection of better common subexpressions.

3.5 Factorization Process

The factorization step of our optimizer is a rule–based process. It requires *factorization rules* and *manipulation rules* to change a LMDFG according to the factorization principles

[4] The anti–join operator is defined as $R \overline{\bowtie} S := \pi_{\mathcal{A}_R}(R \bowtie S)$, where $\mathcal{A}_R$ is the set of attribute names of R.
[5] Two different LMDFG's are equivalent if both produce the same result.

and manipulation principles, respectively. Therefore, the factorization process works as follows: Beginning with an initial LMDFG for the constraints the factorization process searches for the best LMDFG by applying rules successively. To decide which LMDFG is the best one we used a cost model based on estimating the minimal cost of the represented physical execution plans. However, the exact definition of this cost model is beyond the scope of this paper, and is discussed in [Sch95].

Because in many cases a complete search in the space of reachable LMDFGs is not feasible in practice, we decided to split the factorization process into two stages. In the lower *factorization stage* the best LMDFG will be chosen which might be constructed from a given (initial) LMDFG using factorization rules. Notice, that no further factorization rule can be applied to such a LMDFG. Therefore, in the higher *manipulation stage* these LMDFGs will be changed by manipulation rules to give it back to the factorization stage where, maybe, further factorization rules can be used. This switching between the factorization and manipulation stage is continued until neither further factorization nor further manipulation rules can be applied. The search strategy of each stage is independent of the other. Because of that, the factorization amount might be reduced further if it is too expensive. For example, it is to be expected that using best first search for the manipulation stage and complete search for the factorization stage produces better solutions than best first search on both stages. Figure 8 gives a overview of the factorization process.

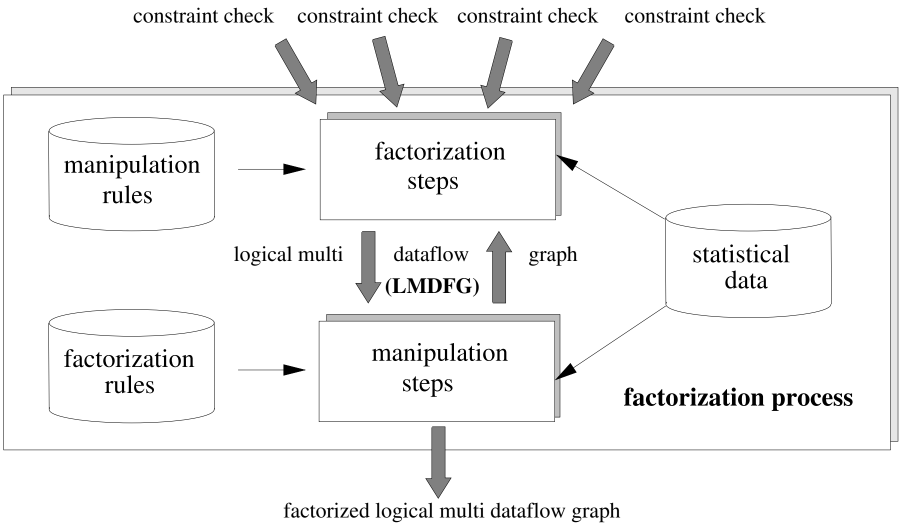

Figure 8: Factorization process overview

Our rule design guarantees termination of the search process. For example, two join nodes will only be merged through the corresponding manipulation rule if one node is the unique successor of the other node. Therefore, never ending cycles of applying the manipulation and factorization rules are impossible. As a consequence of this overall rule characteristic, the search strategies do not need checks for rule cycles.

4 Parallelization of Execution Plans

4.1 Forms of Parallelism

In order to improve the response time for constraint checks on very large databases, we attempt to exploit parallelism. Similar to performance improvement of query processing in a database system we distinguish for efficient constraint evaluation inter–operator–parallelism, and intra–operator–parallelism. In the first one — inter–operator–parallelism — can be achieved by the following two principles: independent paths and pipeling. *Independent paths* of the dataflow graph can be executed in parallel. Two paths are independent if the nodes of each path are not in a producer–consumer relationship. *Pipelined processing* is possible if the consumer does not have to wait until all tuples of the input become available but may already start execution as soon as the first tuple has been received, and the producer is able to generate output tuples step by step.

To exploit intra–operator–parallelism an operator has to be replicated, with the duplicates operating on disjoint input subsets (*node splitting*). Thus each duplicate operator can be executed independently and in parallel with the others. This form of parallelism is also known as horizontal parallelism and is achieved by data distribution.

4.2 Short Description of the Cost Model

The optimization goal for parallel constraint evaluation is the reduction of the execution time (in contrast to execution cost for sequential constraint evaluation). Therefore, we need a metric for the valuation of a constraint execution plan. Such a cost model should include (1) an estimation function for the operators of the constraint execution plan, and (2) an estimation function for the constraint execution plan which combines the individual estimation functions. We compute the following characteristic time points for each operator n: $T_{start}(n)$ — the time of start for the operator, $T_{res}(n)$ — the time of creating the first result tuple of the operator, and $T_{end}(n)$ — the time of creating the last result tuple of the operator. The computation of these time points of an operator is based on the operator costs and on the characteristic time points of its predecessors. Furthermore, we distinguish two phases for each operator: $T_{Phase_1}(n)$ is the minimal time needed by the operator n from $T_{start}(n)$ to $T_{res}(n)$, and $T_{Phase_2}(n)$ is the time to creating the remaining result tuples. Because these time functions are recursively defined on the predecessors, the execution time estimation of the operator graph corresponds to T_{res} of the execute–operator.

4.3 Parallelization Process

Using node splitting: Since pipeline parallelism and parallel execution of independent paths is determined by the shape of the execution plan (and determined in the previous optimized–mapping step), in this step we must choose the *degree of parallelism* of each operator in the constraint execution plan. The assignment of specific processors to execute each operator is left to the execution time and is not considered in this paper. The degree of parallelism of an operator should be selected so that the cost due to the parallelism overhead (starting and terminating of all operator instances, resource contention, etc.) is lower than performance improvement due to parallelism.

Algorithm to determine the degree of parallelism: To improve the performance of the execution plan evaluation, the degree of parallelism of any operator can be increased. To obtain an optimal degree of parallelism for each operator in the execution plan, a brute force algorithm can systematically increase the degree of parallelism by 1. Clearly, this is quite inefficient, and also does not provide a termination criterion. In contrast, our strategy is to consider only those operators which can lead to a reduction of the execution time by a further increase of their degree of parallelism. Such an operator is called *critical operator*. In each execution plan exists operators whose parallelization does not lead to execution time reduction, because they have consumers which cannot consume the results any faster.

Definition 4.1 (critical operator) *A critical operator is an operator of the constraint execution plan. The parallelization of the critical operator can lead to a performance improvement.*

Algorithm 4.2 (determining critical operators $\mathcal{K}$)

1. $\mathcal{K} = \emptyset$ *and*
   $\mathcal{K}' = \{n\}$, *and for* n *holds:*
   $T_{end}(n) = \max\{T_{end}(n') |\ n'$ *is a producer of the execute–operator*$\}$ *and*
   $T_{Phase_2}(n) = T_{end}(n) - T_{res}(n)$
2. ***while*** $\mathcal{K}' \neq \emptyset$ ***do***
   (a) *remove* n *from* $\mathcal{K}'$.
   (b) $\mathcal{K}' = \mathcal{K}' \cup \{node(n.op) |\ n.op$ *is a operand of* $n\}$
   (c) ***if*** $T_{Phase_1}(n) = T_{res}(n) - T_{start}(n)$ ***or*** $T_{Phase_2}(n) = T_{end}(n) - T_{res}(n)$
   then $\mathcal{K} = \mathcal{K} \cup \{n\}$.

The optimizer starts from an initial parallel execution plan. According to algorithm 4.2, the set of critical operators in the execution plan is determined. For each critical operator a new execution plan is created with a degree of parallelism increased by one for the critical operator. After estimating the evaluation costs, each of these execution plans is a starting point for the determination of further critical operators and therefore further parallelization. Search paths are abandoned, if the last $(k-1)$ steps did not achieve an improvement with respect to the best execution plan. This is a sufficient termination criterion, because $(k-1)$ is the number of operators in the execution plan without the execute–operator and therefore, no better plans can be reached on this path. To reduce the optimization time, the search space is saved in order to prevent the repeated generation of search space elements.

5 Performance Evaluation

5.1 Test Data and Integrity Constraints

The examples will be drawn from a case study of a hospital information system. For each patient diagnoses are given by doctors or senior consultants. On the basis of a diagnosis therapies are ordered. Therapies consist of different treatments, e.g. surgery, medication, physiotherapy, and radiotherapy. Treatments are carried out by doctors, senior consultants, nurses, physiotherapists, and other staff. A hospital consists of different wards. Patients are put in rooms within wards. They are looked after by the ward's staff. While staying in the hospital patients get meals. The entity–relationship model of the described miniworld and the corresponding relational schema are given in Appendix A.

The database was generated with our Pace database generator [Sch95]. For our experiments we consider the information in a hospital during a quarter of a year. In the hospital are 500 rooms. In the considered period of time 2000 patients are admitted to the hospital. For these patients 4000 diagnoses are made, and 8000 therapies are ordered. The treatments consist of 10000 medications, 2000 physiotherapies, 5000 radiotherapies, and 1000 surgeries. The complete size for the hospital database is summarized in table 1.

relation	**size**	**relation**	**size**	**relation**	**size**
Patient	2000	Medication	10000	Room	500
Diagnosis	4000	Physiotherapy	2000	Meal	2190
Therapy	8000	Radiotherapy	5000	Surgery	1000
Staff	500	LookAfter	2500	BelongsTo	800
Ward	50	CarryOut	20000	GetMeal	15000

Table 1: Size of the example database

We suppose the following set of integrity constraints, specified in first–order logic[6].

1. Therapies which cost more than 30000 $ and are surgery, chemotherapy, medication, or radiotherapy must be ordered by a senior consultant.
   $\forall$ th_id, cost, st_id Therapy(th_id,cost,*,st_id) $\wedge$ $>$ (cost,30000) $\wedge$ (Surgery(*,*,*,*,th_id) $\vee$ Medication(*,'Chemo',*,*,th_id) $\vee$ Radiotherapy(*,*,*,*,th_id)) $\Longrightarrow$ Staff(st_id,'Senior')

2. Patients do not get a meal on the day of surgery.
   $\forall$ p_id, d_id, th_id, t_id, m_id, date
   Patient(p_id,*) $\wedge$ Diagnosis(d_id,*,p_id,*) $\wedge$ Therapy(th_id,*,d_id,*) $\wedge$
   Surgery(*,*,date,*,th_id) $\Longrightarrow$ $\neg$ (GetMeal(p_id,m_id) $\wedge$ Meal(m_id,date))

[6] Irrelevant variables are marked by a star (*).

3. Patients are looked after by the ward's nurses.
$\forall$ p_id, p_class, r_id, st_id $\exists$ w_id Patient(p_id,p_class) $\wedge$ LookAfter(p_id,r_id,st_id,*) $\Longrightarrow$ Room(r_id,w_id,*,*) $\wedge$ Staff(st_id,'Nurse') $\wedge$ BelongsTo(st_id,w_id)

4. Private patients are put only in private rooms.
$\forall$ p_id, r_id, Patient(p_id,'Private') $\wedge$ LookAfter(p_id,r_id,*,*) $\Longrightarrow$ Room(r_id,*,*,'Private')

5. Surgeries of private patients must be carried out by a senior consultant.
$\forall$ p_id,d_id,th_id,t_id $\exists$ st_id
Patient(p_id,'Private') $\wedge$ Diagnosis(d_id,*,p_id,*) $\wedge$ Therapy(th_id,*,d_id,*) $\wedge$
Surgery(t_id,*,*,*,th_id) $\Longrightarrow$ CarryOut(st_id,t_id) $\wedge$ Staff(st_id,'Senior')

6. Patients with serious surgeries must be looked after at the day of surgery.
$\forall$ p_id,d_id,th_id, date $\exists$ r_id, st_id Patient(p_id,*) $\wedge$ Diagnosis(d_id,*,p_id,*) $\wedge$
Therapy(th_id,*,d_id,*) $\wedge$ Surgery(*,*,date,'Serious'*,th_id) $\Longrightarrow$ LookAfter(p_id,r_id,st_id,date)

5.2 Experimental Setup

Our prototype runs on a KSR1–8 multiprocessor system. The important feature of the KSR architecture is the virtual shared–memory, so the machine logically provides a shared–memory programming model.

In our experiments we used constraint execution plans generated from the constraints given above. We varied the number of processing nodes maintaining a constant degree of parallelism. Because not all processing nodes have a local disk, all required relations were first loaded into main memory. Thus, the experiments do not include I/O–time.

For performance evaluation we varied the following parameters:

1. First, we show the usefulness of parallelism in checking integrity constraints with the ICE. For that matter, we vary in the experiments the number of available processors for the constraint checks, i.e. we increase the available resources from 1 to 8.

2. The experiments also compare the execution time of the different evaluation plans (with and without factorization, optimized mapping and parallelization) to show the impact of the optimization techniques.

We use the constraints from section 5.1 in the following compositions: constraints 2 and 5, constraints 2 and 6, constraints 1 and 2, and constraints 3 and 4. For each constraint composition we used our Pace plan generator, which compiles constraints from the extended relational algebra into constraint evaluation plans (CEPs). We generate CEPs without optimization (noopt), with factorization (fact) and with factorization and parallelization (fact+par).

5.3 Experimental Results

The figures show the runtime of the constraint checks depending on the number of available processors.

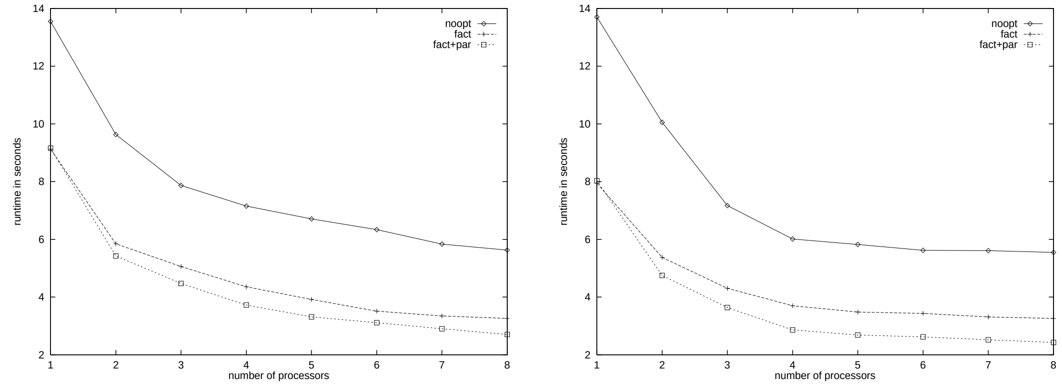

Figure 9: Results: check of constraints 2+5 and of constraints 2+6

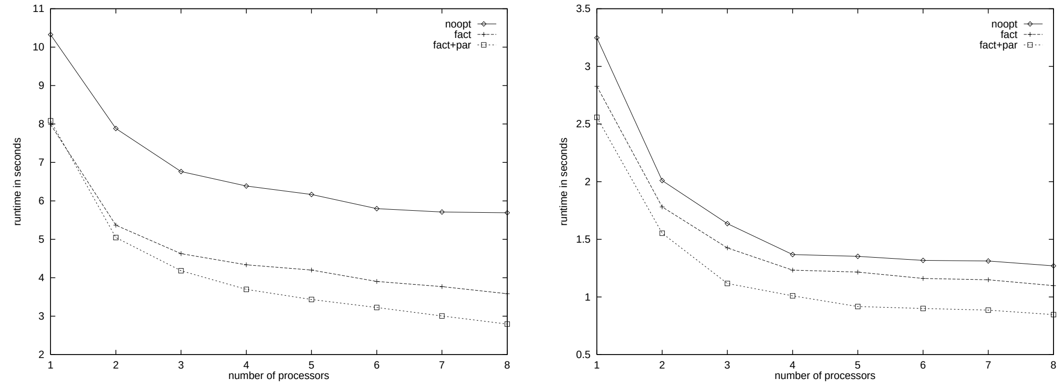

Figure 10: Results: check of constraints 1+2 and of constraints 3+4

5.4 Evaluation and Discussion

The measurements show that it is worthwhile to use parallelism for constraint execution. The runtime of each constraint execution is decreasing with an increasing number of processing nodes. The reason for this effect is, that the inherent parallelism through pipelining and independent operators in the constraint execution plan is supported by our integrity constraint engine [HS95]. The other observation is that factorization leads to a considerable performance improvement. The second optimization technique — parallelization, in conjunction with optimized mapping — further improves the efficiency of the constraint checks if sufficient resources are available, i.e. the number of processors is greater than two. We expect that when we use more than 8 processors, the difference between plans using factorization and factorization with parallelization will be even more significant.

6 Conclusions and Future Work

We have presented two techniques for checking integrity constraints efficiently. The underlying idea is to treat integrity constraints as relational queries. This approach supports the specification of constraints in a declarative way. For efficient treatment of the complex integrity constraints, we started from a relational algebra with generalized join operators. The algebraic expressions are subject to optimization. This is not unlike approaches for relational query optimization, although we use a compile–time rather than a query–execution–time approach. Therefore, the simplification and optimization can take place during the definition of integrity constraints.

Because we have to check many constraints simultaneously, in a second step common subexpression are identified and factorized during global optimization. Similar to approaches in distributed and parallel databases, the algebraic expressions also serve as a basis for parallelization to obtain further gains in performance. However, a main difference to other approaches is the determination of the final operator ordering according to a parallel cost model. Other works often only consider the best sequential plan as a starting point for the parallelization. Due to the prior factorization, we are considering a LMDFG as input for the optimized mapping and therefore parallelization. Such a LMDFG does not fix the ordering of subsequent operators of the same type and therefore, represents a collection of different plans which are our starting points for the parallelization. We have integrated the determination of operator ordering into the search for the best parallel execution plan. The resulting plan is executed by a parallel integrity constraint engine [HS95]. The experiments on a parallel KSR1 machine showed that global optimization and parallelism can significantly reduce the runtime of constraint checks in comparison to sequential evaluation.

In the future we plan to address the following issues:

- Up to now, we ignored the problems of load sharing and load balancing in the hope that the operating system can do this job properly. However, in our experiments we recognized that it is not always adequate to leave this task to the operating system. Consequently, we should incorporate the optimal initial operator–node–mapping (load sharing) and the reorganization of the thread allocation (load balancing) into the ICE.

- In the factorization step as well as in the parallelization step the search spaces can be very large. As an next step we will consider heuristic search strategies like best–first search or randomized search strategies like iterative improvement and simulated annealing [Ree93] in order to reduce the optimization time.

References

[AR94] J.R. Alsabbagh and V.V. Raghavan. Analysis of common subexpression exploitation models in multiple query processing. In *Proceedings of the 10th International Conference on Data Engineering*, pages 488–497, Houston, TX, February 1994.

[CLS93] A. Cosar, E.P. Lim, and J. Srivastava. Multiple query optimization with depth-first branch-and-bound and dynamic query ordering. In *Proceedings of the 2nd International Conference on Information and Knowledge Management*, pages 433–438, Washington, DC, 1993.

[DG85] D.J. DeWitt and R. Gerber. Multiprocessor hash-based join algorithms. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 151–164, Stockholm, Sweden, 1985.

[DG92] D.J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[GA93] P.W.P.J. Grefen and P.M.G. Apers. Integrity control in relational database systems – An overview. *Data & Knowledge Engineering*, 10(2):187–223, 1993.

[GFA92] P.W.P.J. Grefen, J. Flokstra, and P.M.G. Apers. Performance evaluation of integrity control in a parallel main-memory database system. In *Proceedings of the 3rd International Conference on Database and Expert Systems Applications*, Valencia, Spain, 1992.

[HM94] U. Herzog and G. Moerkotte. Eine Basis für effiziente Konsistenzprüfung. Technical Report 7/94, Universität Karlsruhe, Fakultät für Informatik, December 1994. In German.

[HS91] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. In *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems*, pages 218–225, Miami Beach, FL, December 1991.

[HS95] U. Herzog and R. Schaarschmidt. Parallel execution of integrity constraint checks. In *Proceedings of the 4th International Conference on Information and Knowledge Management*, Baltimore, ML, November 1995.

[MK94] N. McCarroll and J. Kerridge. A strategy for semantic integrity enforcement in a parallel database machine. In *Proceedings of the 12th British National Conference on Databases*, pages 137–152, Guildford, UK, July 1994.

[Nic82] J.M. Nicolas. Logic for improving integrity checking in relational data bases. *Acta Informatica*, 18(3):227–253, 1982.

[PS88] J. Park and A. Segev. Using common subexpressions to optimize multiple queries. In *Proceedings of the 4th International Conference on Data Engineering*, pages 311–319, February 1988.

[Ree93] C.R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems*. Blackwell, 1993.

[Sch95] J. Schlösser. Generierung paralleler Bearbeitungspläne für Konsistenztests. Master's thesis, Universität Karlsruhe, Fakultät für Informatik, January 1995. In German.

[SE93] J. Srivastava and G. Elsesser. Optimizing multi-join queries in parallel relational databases. In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, pages 84–92, San Diego, CA, 1993.

[Sel88] T.K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, March 1988.

[SG90] T. Sellis and S. Ghosh. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):262–266, June 1990.

[SSN94] K. Shim, T. Sellis, and D. Nau. Improvements on a heuristic algorithm for multiple-query optimization. *Data & Knowledge Engineering*, 12:197–222, 1994.

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

A Database Schema

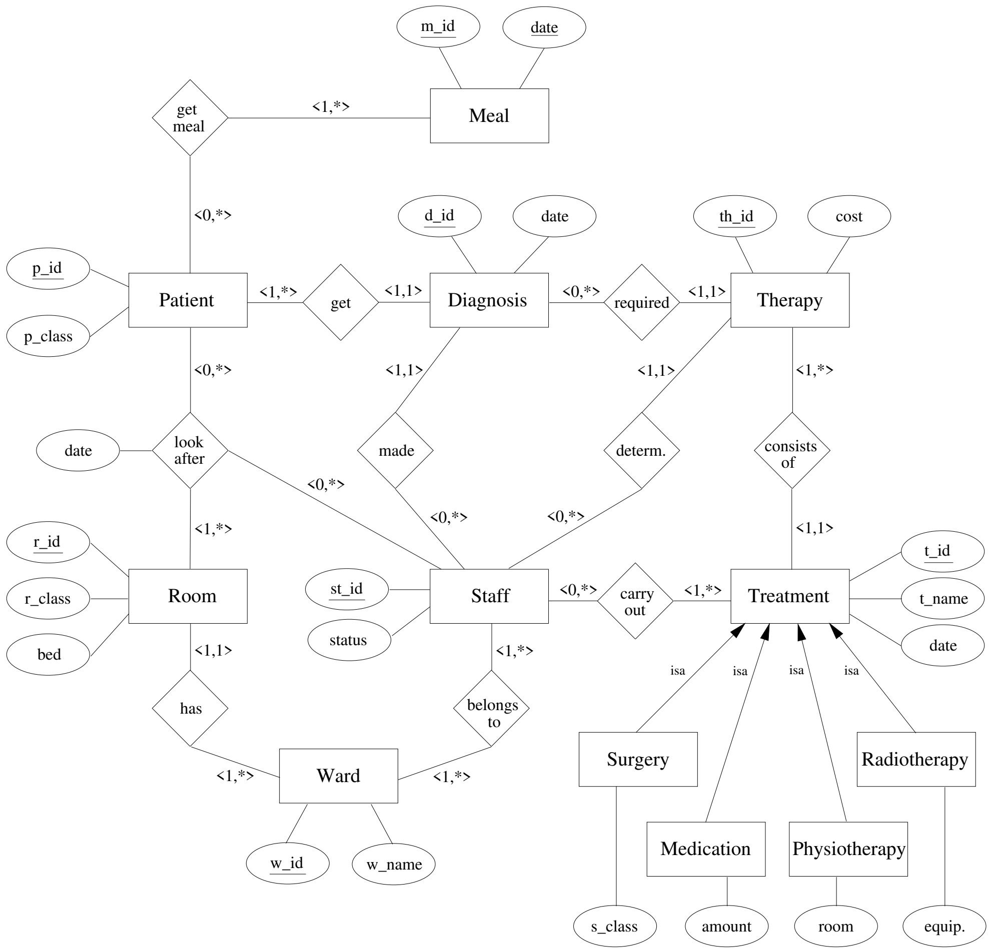

Figure 11: Entity-Relationship Model of the hospital information system (part)

Using standard rules figure 11 is translated to a (simplified) database schema containing the following relations. Key attributes are underlined:

relation Patient(p_id,p_class)
relation Therapy(th_id,cost,d_id,st_id)
relation Ward(w_id,w_name)
relation Physiotherapy(t_id,t_name,date,room,th_id)
relation Medication(t_id,t_name,date,amount,th_id)
relation Radiotherapy(t_id,t_name,date,equip,th_id)
relation CarryOut(st_id,t_id)
relation GetMeal(p_id,m_id)
relation Diagnosis(d_id,date,p_id,st_id)
relation Staff(st_id,status)
relation Room(r_id,w_id,bed,r_class)
relation Surgery(t_id,t_name,date,s_class,th_id)
relation Meal(m_id,date)
relation LookAfter(p_id,r_id,st_id,date)
relation BelongsTo(st_id,w_id)