An Optimal Predicate Locking Scheduler

Carlo Meghini and Costantino Thanos
Istituto di Elaborazione della Informazione
Consiglio Nazionale delle Ricerche
Via Santa Maria 46, I-56100 Pisa, Italy
e-mail: {meghini,thanos}@iei.pi.cnr.it

Abstract

The paper presents a predicate locking scheduler that maximizes concurrency by locking as many of the database entities as possible without compromising the correctness of execution of the database transactions. The scheduling strategy that guarantees the maximal concurrency is first identified, then a predicate language allowing an efficient implementation of this strategy is given. The optimal predicate locking scheduler is successively presented, based on a lattice-theoretic formalization of the underlying concepts. Finally, the range of applicability of the optimal scheduling strategy is circumscribed, by showing that any significant extension to the expressive power of the predicate language accepted by the optimal scheduler causes an irreparable loss of efficiency.

1 Introduction

There is a vast literature on the theory and practice of database concurrency control [11, 1]. A number of algorithms for concurrency control protocols have been proposed, implemented and used in commercial systems. The most common technique for concurrency control is locking. One way to use this technique to obtain correct schedules is to adopt two-phase locking (2PL) [6]. It is interesting to note that 2PL, which was historically the first concurrency control technique to be proposed, is in some sense the best. It can be shown that, when transactions access database entities without any specific order, they must be two-phase locked to preserve the integrity of the database. From a different point of view, [11] proved that, when the concurrency control algorithm has only dynamic information on database entities, then the greatest concurrency achievable is that of 2PL.

When concurrency control is considered, the database is modelled as a fixed set of entities which can be accessed by read and write operations. However,

real databases can grow and shrink dynamically. In addition to read and write, they support insert and delete operations. In the case of dynamic databases a concurrency control problem arises: the phantom problem [1]. This problem causes a transaction to obtain inconsistent views of the database, due to the appearing and disappearing of some database entities, like ghosts. In this case, 2PL does not guarantee the correct execution of concurrent database transactions. In [1], some techniques to solve the phantom problem are discussed.

A more general technique to solve this problem is predicate locking. Predicate locking allows sets of database entities rather than single entities to be locked. This is obtained by using complex predicates, such as Boolean combinations of simple predicates, to denote sets of database entities.

Predicate locking is not widely used because it poses two main problems. First, it is very expensive, since it requires the detection of conflicts between predicates. Second, it performs poorly, since it permits a low level of concurrency between concurrent database transactions.

In this paper we present a predicate locking scheduler that is both optimal and efficient. The former property means that the scheduler maximizes concurrency. The latter means that, by limiting the expressive power of the predicate language, the scheduler can be realized by a polynomially bounded algorithm. Since we also show that any significant extension to the predicate language causes the scheduler to lose its efficiency, our scheduler turns out to be the best predicate locking scheduler that maximizes concurrency.

The paper is structured as follows. Section 2 presents a formal model of predicate locking, and defines a correctness criterion for predicate locking schedulers. The typical predicate locking scheduler is briefly examined in Section 3, with the aim of showing its limits with respect to concurrency. In Section 4, the scheduling strategy guaranteeing maximal concurrency is presented and proved to be correct. Section 5 provides an efficient implementation of this strategy, identifying a predicate language that allows the effective computation of the predicates involved in the strategy. Finally, it is shown that any significant extension to the expressivity of the identified predicate language results in the loss of the efficiency of the scheduler.

2 The model

We present a model that extends the model for database concurrency control presented in [11] to the predicate locking case.

2.1 The database and the predicate language

Most of the work on predicate locking (for example [6, 12, 2]) assumes a record-based database model, quite often the relational model, in which expressions of the tuple calculus [3] are used as predicates. Our aim is to define a model that, following [11], makes no assumptions on the structure of the objects of the database, and allows the denotation of sets of such objects by means of the formulae of a first-order language.

Definition 1: A *database* is a triple $\langle E, \mathcal{D}, \mathcal{S} \rangle$ where

(i) $E = \{e_1, e_2, \ldots\}$ is a countable set of *database entities;*

(ii) $\mathcal{D} = \{\langle D_1, \prec_1 \rangle, \langle D_2, \prec_2 \rangle, \ldots, \langle D_m, \prec_m \rangle\}$, $m \geq 1$, is a set of *domains* of the database; each D_i is a non-empty set, totally ordered by the binary relation $\prec_i$;

(iii) $\mathcal{S} = \{\langle N_1, R_1 \rangle, \langle N_2, R_2 \rangle, \ldots, \langle N_n, R_n \rangle\}$, $n \geq m$, is a set of *properties* of the database, where for all $1 \leq i \leq n$, N_i is a name, and R_i is either E, in which case $\langle N_i, R_i \rangle$ is said to be a *complex* property, or R_i is D_j for some $1 \leq j \leq m$, in which case $\langle N_i, R_i \rangle$ is said to be a *simple* property.

A *database state DB* is a pair $\langle E_{DB}, F_{DB} \rangle$ where E_{DB}, the *state entities,* is a finite subset of E, and F_{DB} is a mapping assigning to each property name a total function from the state entities to the property range, that is, for all $1 \leq j \leq n$, $F_{DB}(N_j) : E_{DB} \rightarrow R_j$ if N_j is the name of a simple property, and $F_{DB}(N_j) : E_{DB} \rightarrow E_{DB}$ if N_j is the name of a complex property. □

The database entities are abstract objects which occur in database states; a database state gives a status, i.e. a set of property values, to its entities, consistently with the information contained in the database schema. The model allows entity aggregation [15] by letting complex properties range over database entities. Simple properties range over totally ordered sets, the domains, which in commercial database systems are numbers, time values, strings of characters, and so on, each of which is equipped with a total ordering relation.

The notion of database state permits us to model a database as an object that evolves over time, and whose entities can be denoted by means of expressions involving their properties, such as those of the language introduced by the next definition.

Definition 2: Given a database $\langle E, \mathcal{D}, \mathcal{S} \rangle$, the *database predicate language* $\mathcal{L}$ is the many-sorted first-order language defined as follows:

(i) the *sorts* of $\mathcal{L}$ are $S_E, S_1, S_2, \ldots, S_m$; the *alphabet* of $\mathcal{L}$ consists of the following symbols: one *constant* symbol of sort S_E for each database

entity, and, for all $1 \leq i \leq m$, one constant symbol of sort S_i for each element in the domain D_i; countably many *variable* symbols $x, y, z, \ldots$, all of sort S_E; for all $1 \leq i \leq n$, one *function* symbol N_i of sort $\langle S_E, S_E \rangle$ if N_i is a complex property name, whereas if N_i is the name of a simple property ranging over D_j, the sort of N_i is $\langle S_E, S_j \rangle$; for all $1 \leq i \leq m$, one binary *predicate* symbol $\prec_i$ of sort $\langle S_i, S_i \rangle$, and the equality predicate symbol, which for short we use in place of $(m+1)$ equality symbols, one for each sort S, of sort$\langle S, S \rangle$;

(ii) for any sort S, a *constant term* of sort S is a constant symbol of sort S; an *atomic function term* of sort S is an expression of the form $N_i(x)$, where x is a variable and N_i is of sort $\langle S_E, S \rangle$; a *function term* of sort S is either an atomic function term of sort S, or is an expression of the form $N_i(t)$, where t is a function term of sort S_E and N_i is of sort $\langle S_E, S \rangle$;

(iii) a *simple atomic formula* has the form $\gamma(\phi, \delta)$, where γ is a predicate symbol of sort $\langle S, S \rangle$, ϕ is a function term of sort S, and δ a constant term of sort S; a *complex atomic formula* is an expression of the form $\gamma(\phi_1, \phi_2)$, where γ is as above and both ϕ_1 and ϕ_2 are function terms of sort S;

(iv) the *well-formed formulae* are built out of the atomic formulae by using the logical connectives $\neg, \wedge$ and quantifier $\exists$ in the standard way. □

As is customary, we will assume all the other typical logical connectives, such as $\vee, \supset$ and $\equiv$, and the universal quantifier $\forall$ as part of our language, introduced as abbreviations of the corresponding official expressions.

Example 1: Let us consider a database regarding people's social life, with the properties:

$$\langle Best\_friend, E \rangle, \langle Lives\_in, String \rangle, \langle Hobby, String \rangle, \langle Age, PositiveInteger \rangle.$$

In order to know who is not an amateur musician and lives where her best friend lives, the following query can be formulated:

$$Hobby(x) \neq Music \wedge Lives\_in(x) = Lives\_in(Best\_friend(x)).$$

In order to simplify notation, the variable specification in function terms will be omitted, when no ambiguity can arise. The people who are younger than their best friend are found by the query:

$$Age \leq Age(Best\_friend).$$ □

Table I: A database state

x	$Best\_friend(x)$	$Lives\_in(x)$	$Hobby(x)$	$Age(x)$
e_1	e_2	Pisa	Tennis	27
e_2	e_3	Pisa	Cooking	30
e_3	e_1	Florence	Music	32

A database is an interpretation of the constant and predicate symbols of its predicate language, because, assigning the usual meaning to equality, it provides an extension for these symbols. In the same way, a database state is an interpretation of the function symbols of the database language. Therefore, by means of the standard semantics for first-order languages [5], sets of database entities can be associated via a database state to open formulae of the database language with one free variable. This is introduced in the model by the next definition, where we assume as known the notions of free variable and truth in a first-order language.

Definition 3: Given a database $\langle E, \mathcal{D}, \mathcal{S} \rangle$ and its predicate language $\mathcal{L}$, let $\mathcal{L}_1$ be the subset of $\mathcal{L}$ consisting of the open formulae of $\mathcal{L}$ with one free variable, also called *predicates.* Given a database state DB and a predicate $\phi(x)$, the *extension of* $\phi(x)$ *in DB,* $\varepsilon_{DB}[\phi(x)]$, is the set of entities $e \in E_{DB}$ such that $\phi(e)$ is true in DB. □

The extension of the first query of the previous example in the database state shown in Table I is the set $\{e_1\}$, whereas the extension of the second query is $\{e_1, e_2\}$.

In the rest of the paper, we will assume a database consisting of one domain, the set of natural numbers, totally ordered by the arithmetical relation $\leq$. As it will be shown, this assumption is not strictly needed by our scheduler, which works also for arbitrarily dense domains, but is dictated by simplicity. The predicate language of our database has one constant symbol for each natural number, and no others, and just one binary predicate symbol beside equality, standing for $\leq$. To simplify notation, we assume the natural numbers as constant symbols and $\leq$ and $=$ as predicate symbols. We will omit the specification of this database whenever no ambiguity can arise.

2.2 Steps, Transactions, Schedules

A step is the atomic unit of interaction between the user and the database. In predicate locking, steps are lock or unlock actions on sets of database entities

denoted via predicates, which come into existence at certain time points. Other kinds of actions, such as read or write, play no role in assessing whether or not a schedule is legal, therefore they are ignored in the model.

Definition 4: Let T be a totally ordered countable set of time points. A *step* is an element of a countable set S, on which the following total functions are defined:

$a : S \rightarrow \{lock, unlock\}$, the *action* of the step;

$p : S \rightarrow \mathcal{L}_1$, the *predicate* of the step; and

$t : S \rightarrow T$, the *time* of the step. □

We call a lock (unlock) step a step whose action is *lock* (*unlock*). A lock step s represents a lock request on the entities that, in the current database state, are denoted by the predicate $p(s)$. Unlock steps, although structurally similar to lock steps, have quite a different meaning: one such steps s has to be understood as the request to release the entities which have been granted to a previously output lock step s' with the same predicate as s. In fact, unlock steps correspond one-to-one with output lock steps, and the correspondence is established via their predicate. The rationale behind this interpretation of unlock steps is that the status of an entity may be changed by the transaction that has locked it; as a result, that entity could no longer satisfy the predicate of the step that is supposed to unlock it, thus remaining locked after the completion of the transaction that has used it. As the new status of the locked entities will in general be unpredictable, it is unrealistic to ask transactions to specify the correct unlock predicates, hence the unlock steps have to be interpreted as described above.

The *time* value of an input step is intended as the moment in which the request is received by the scheduler, whereas the *time* value of an output step is intended as the moment in which the scheduler outputs the step. It is thus reasonable to assume, as we will, that no two steps can have the same time. A step s is said to reference an entity e in a database state DB when e belongs to the extension of *p(s)* in *DB.*

In the classical theory of concurrency control, a lock step applies to a single database entity, specified as argument of the step; two lock steps are said to *conflict* if they apply to the same entity. In predicate locking, two important differencies exist: first, lock steps reference set of entities; second, they do it in an indirect way, via predicates that denote these entities without explicitly mentioning them. The former fact is dealt with by defining conflict in terms of the non-disjointness of the involved sets of entities. The latter fact requires

considering database states, as the entities denoted by a predicate vary in accordance with the database state. As a result, two lock steps may address a common entity in a state but not in another one. The problem is solved by taking a very conservative approach, illustrated by the next definition.

Definition 5: Two lock steps s_1 and s_2 conflict if there exists a database state DB in which they reference a common entity, i.e. $\varepsilon_{DB}[(p(s_1)] \cap \varepsilon_{DB}[(p(s_2))] \neq \emptyset$. □

It is important to observe that this notion of conflict avoids the phantom problem mentioned in the Introduction. In addition, it can be stated in logical terms.

Proposition 1: Two steps s_1 and s_2 conflict if and only if $(p(s_1) \wedge p(s_2))$ is satisfiable.
Proof: $(p(s_1) \wedge p(s_2))$ is satisfiable if and only if there exists a database state DB such that $\varepsilon_{DB}[(p(s_1) \wedge p(s_2))] \neq \emptyset$, or, equivalently, $\varepsilon_{DB}[(p(s_1)] \cap \varepsilon_{DB}[(p(s_2))] \neq \emptyset$. □

Sequences of lock and unlock steps constitute locked transactions, according to well known rules [11]. These rules can be imported in the predicate locking case as follows.

Definition 6: A *locked transaction,* or simply a *transaction, LT*, is a sequence of steps satisfying the following conditions, for each database state DB and database entity $e \in E_{DB}$:

(i) there is at most one lock step s in LT referencing e;

(ii) a lock step referencing e is in LT if and only if there is in LT exactly one unlock step referencing e;

(iii) a lock step referencing e in LT precedes an unlock step referencing e in LT;

(iv) the order of the steps is consistent with the time of the steps, that is a step s_i precedes a step s_j in LT if and only if $t(s_i) < t(s_j)$. □

This definition guarantees that the predicates of the steps of transactions are globally equivalent, ruling out: (a) transactions that "forget" to unlock previously locked items, such as $T : s_1 s_2$, where s_1 is a lock step with predicate $A \geq 0$ and s_2 is an unlock step with predicate $A \geq 1$; and (b) transactions that unlock more data than previously locked, such as T as defined above except that s_1's predicate is $A \geq 2$. The definition is based on a static criterion, assuming that the database state will be the same during the execution of a transaction.

The special semantics given to unlock steps guarantees its correctness also in case of state changes. It should also be noticed that the set of transactions is not decidable, because, as has just been shown, testing whether two steps reference the same database entity amounts to testing the satisfiability of a first-order formula, a problem known to be unsolvable. The issue will be taken up later, when the predicate language will be restricted in order to make the satisfiability test effective.

Schedules are formed by mixing the steps of transactions.

Definition 7: A *schedule* of the transactions $LT_1, LT_2, \ldots, LT_k$ is a function that associates a set S of steps to the k-tuple $\langle LT_1, LT_2, \ldots, LT_k \rangle$ such that a step is in S if and only if it is in one of $LT_1, LT_2, \ldots, LT_k$. □

A schedule, and in general any set of steps, can be seen as a sequence of steps by ordering the steps in the set by their occurrence in time; in the rest of the paper we will sometimes treat sets of steps as sets and sometimes as sequences, depending on which of the two is most convenient in the circumstance.

2.3 Predicate Locking Schedulers

A scheduler transforms schedules into schedules, thus it is most naturally seen as a function whose domain and range are schedules. However, not all schedules can be reasonably given as input to a scheduler. Consider the transactions a_1a_2 and b_1b_2, where a_1 and b_1 are lock steps whose corresponding unlock steps are a_2 and b_2, respectively. In addition, assume that a_1 and b_1 conflict. Now consider a scheduler that adopts a first-in first-out policy, that is it outputs first the lock steps it receives first (provided that the output schedule is correct, of course). Then the sequence $a_1b_1b_2a_2$, which according to the above definition is a schedule, cannot be considered a realistic input to this scheduler. For the scheduler would grant the lock requested by a_1, defer the step b_1 to preserve the correctness of the output, and then be faced with step b_2, asking to unlock entities that have not yet been granted. If, on the other hand, the scheduler was designed in such a way that it would delay a_1 and grant b_1, then the sequence $a_1b_1a_2b_2$ could not be accepted as a reasonable input schedule.

As it turns out, if we want to define schedules as the sequences of steps that could realistically be given as input to a scheduler, we ought to define schedules in terms of schedulers. On the other hand, schedulers being functions on schedules, they should be defined in terms of schedules. The way out of this circularity is to consider schedulers as *partial* functions which are undefined on unrealistic schedules, that is schedules suffering from the anomaly illustrated in the previous example. This partiality has no practical effect on schedulers,

because schedules are generated by the effective interaction between the users and the database, therefore it will never be the case that a user releases entities that have never been received. Consequently, unrealistic schedules will never be submitted to a scheduler.

In the theory of database concurrency control [11], any schedule output by a scheduler must satisfy two conditions: first, it must consist of the same steps as the input schedule; second, it must be legal, that is no two transactions in it may simultaneously hold the same database entity in lock. In predicate locking, the first condition turns out to be too restrictive, because steps reference sets of entities, and forcing the output lock steps to be the same as the input ones would make the scheduler handle the database entities in rigidly defined packages, decided by the users. Our approach, presented in detail in Section 4.1, is based on the splitting of each input lock step in two main sub-steps: one addressing only entities held by some other transaction, the other addressing only free entities. The former sub-step is delayed, while the latter is output.

The resulting scheduler grants as many as possible of the requested entities, having a much more active role than the classical scheduler. While the latter simply says a "yes" or a "no" on every input step, the former analyzes the input lock steps and decides what entities to grant, on the basis of an optimality criterion. But it is important to observe that this higher flexibility of the scheduler produces the desired increased concurrency only in presence of a higher flexibility of transactions. In particular, transactions must satisfy two requirements. First, since the requested data are granted by the scheduler in successive batches, the transactions must be able to process the data in successive batches. Many applications fit into this scenario, for instance all those requiring sequential processing of the data. The added programming complexity seems affordable, whether it is given to the application coder or it is taken by the system, which makes the data fragmentation entirely transparent as in distributed query processing. Second, the data must be released as soon as possible and in the same format they have been received. This means that the input unlock steps strictly correspond to the output lock steps, as the latter establish how the requested data are granted. If any of these two requirements is missed, then there is no gain in adopting such a sophisticated scheduler. However, there is also no loss.

In order to render our model adequate to the just described splitting strategy, we will require that for each possible database state *DB,* input step s and entity e referenced by s in*DB,* there must be exactly one output step referencing e in *DB,* and vice versa. Given our assumptions on unlock steps, and in order to introduce time, we further refine the condition in question to the following;

for any database state *DB:*

(1) there exists a total, injective function that associates to each input lock step s and entity e referenced by s, a lock step s' referencing e output by the scheduler no earlier than s;

(2) there exists a total, injective function that associates to each output lock step s and entity e referenced by s, a lock step s' referencing e input to the scheduler no later than s;

(3) an unlock step is output by the scheduler if and only if it is input to the scheduler.

As far as the legality of the output schedules is concerned, by definition a transaction never asks to lock the same entity twice, therefore conflicts always arise between steps of different transactions. We can then state the legality condition as follows: for any database state *DB* and entity e,

(4) if two lock steps referencing e are output, then an unlock step referencing e is output in between them.

In order to define schedulers, the following abbreviations are introduced; for a database state *DB,* a set of steps A, an entity e, and a time point t :

$$\begin{aligned} LS(A) &= \{(l,e) \mid l \in A,\ a(l) = lock,\ e \in \varepsilon_{DB}[p(l)]\}, \\ U(A) &= \{u \in A \mid a(u) = unlock\}, \\ C(A,e,t) &= \sum_{s \in A} count(s,e,t) \end{aligned}$$

where

$$count(s,e,t) = \begin{cases} 0 & \text{if } t(s) > t \\ -1 & \text{if } t(s) \leq t,\ e \in \varepsilon_{DB}[p(s)] \text{ and } a(s) = unlock \\ 1 & \text{if } t(s) \leq t,\ e \in \varepsilon_{DB}[p(s)] \text{ and } a(s) = lock \end{cases}$$

In practice, $LS(A)$ contains the pairs (lock step, requested entity) occurring in a given set of steps A; $U(A)$ are the unlock steps in A; $C(A,e,t)$ gives the balance of a database entity e in the set A at time t, lock steps being counted positively, unlock steps negatively.

Definition 8: A scheduler F is a partial function from schedules to schedules such that for all database states and schedules $sch \in dom(F)$:

(i) there is a one-to-one mapping R between $LS(sch)$ and $LS(F(sch))$, such that
$((l,e),(l',e')) \in R$ if and only if $e = e'$ and $t(l) \leq t(l')$;

(ii) $U(sch) = U(F(sch))$;

(iii) for any database state, entity e and time point t, $C(F(sch), e, t) \leq 1$. □

The first condition is the formal counterpart of sentences (1) and (2), the second of sentence (3), and the third of sentence (4) above. At this point, we notice that any bijection between $LS(sch)$ and $LS(F(sch))$ would capture the same correctness criterion as condition (i); however, we have preferred to follow strictly the informal statements, which come directly from our intuition.

In the previous definition we have in fact abused terminology, because the term "scheduler", which for us denotes a function, is typically used to name the programs which are responsible for concurrency control. From now on we shall return to this latter use of the term, intepreting the conditions given in Definition 8 as requirements for the correctness of these programs.

A useful notion in predicate locking is that of active lock step, that is a step referencing database entities that are currently held by some transactions.

Definition 9: Given a scheduler F and a schedule $sch \in dom(F)$, a lock step s is said to be *active* in F at time t on a database state *DB,* if for each entity $e \in \varepsilon_{DB}[p(s)]$, $count(s, e, t) = 1$. □

3 The typical predicate locking scheduler

Figure 1 presents what can be considered as the typical predicate locking scheduler. The behavior of this scheduler can be characterized as follows: each schedule output by *typ_sch* consists of the same steps as the input schedule; an input lock step s is output by *typ_sch* if and only if it does not conflict with an active lock step; otherwise, s is enqueued and kept in the queue until all the entities it references are available.

In order to realize this behavior, *typ_sch* uses the global variables of type set L and Q to keep track of the active and the queued lock steps, respectively. Both these variables are externally initialized to the empty set. In addition, the Boolean variable *disjoint* records whether the predicate of the step being scheduled conflicts with the predicate of some active lock step. In scheduling a lock step s, *typ_sch* examines the active lock steps in order to ascertain whether one of these steps conflicts with s. This is done by testing the satisfiability of the conjunction of $p(s)$ and the predicate associated to each active step s'. If a conflicting step is found in L, s is queued, that is added to Q. If no step in L conflicts with s, $disjoint$ is $true$: in this case s is added to L and output. When an unlock step s is given as input to *typ_sch*, the corresponding lock is

```
procedure typ_sch (s : step)
s': step
disjoint: boolean
begin
if action(s) = 'lock' then
  begin
  disjoint ← true
  for each s' ∈ L do
      if satisfiable(p(s') ∧ p(s)) then
        begin
        Q ← Q ∪ {s}
        disjoint ← false
        end
  if disjoint then
      begin
      L ← L ∪ {s}
      output(s)
      end
  end
if action(s) = 'unlock' then
  begin
  output(s)
  for each s' ∈ L do
      if p(s') = p(s) then L ← L − {s'}
  for each s' ∈ Q do typ_sch(s')
  end
end
```

Figure 1: The typical predicate locking scheduler

removed from L, s is output, and each queued step is scheduled again to see if it can be granted.

The proof of the following proposition is trivial.

Proposition 2: *typ_sch* is correct. □

The typical scheduler suffers from an evident drawback, caused by the rigidity of its locking strategy, as shown by the following example.

Example 2: Consider a schedule $s_1 s_2 \ldots$, where s_1 and s_2 are lock steps such that $p(s_1)$ is $(1 \leq N_1 \leq 10)$, and $p(s_2)$ is $(N_1 \geq 9)$. Suppose this schedule is given as input to our typical scheduler; in response, the scheduler outputs s_1; then, the conjunction of $p(s_1)$ and $p(s_2)$ being the satisfiable predicate $(9 \leq N_1 \leq 10)$, the scheduler delays s_2 until it receives the unlock corresponding to s_1. In so doing, the scheduler achieves a low degree of parallelism, as it does not grant the lock on the entities satisfying the predicate $(N_1 \geq 11)$, which are *not* locked. A more flexible scheduler would only delay the lock of the already locked entities, in our case those satisfying the predicate $(9 \leq N_1 \leq 10)$, thus deferring the execution of only a portion of the action asking for the lock. □

4 The optimal predicate locking scheduler

The strategy adopted by the typical scheduler is motivated by the fact that computing predicates is in general a difficult task, plagued by very well-known intractability results. In fact, the trade-off between efficiency and concurrency, which is typical of schedulers, is resolved by the typical scheduler in a drastic way: in order to maximize efficiency, it minimizes concurrency. We place ourselves at the other extreme of the trade-off, and give priority to the concurrency of the scheduler. For this purpose, in this section we tackle two problems: first, the identification of the optimal predicate locking scheduler, that is the one allowing the highest degree of parallelism regardless of the database predicate language; second, the study of the performance of such a scheduler, as a necessary step towards the identification of a predicate language allowing tractable operations on predicates.

4.1 The optimal splitting strategy

The basic principle which must guide the operations of the optimal scheduler is very simple: whenever a lock step on a set A of database entities is received, the scheduler should identify the subset of A consisting of the entities not currently held by some other step, and grant a lock on these entities, while delaying granting the others. This will result in a split of the step being scheduled into

sub-steps, so that the largest possible set of database entities is granted, hence the name of *optimal splitting strategy* for this scheduling policy. Upon scheduling an unlock request, the scheduler should act in a similar way, granting as many as possible of the unlocked entities to the pending requests. A somewhat similar strategy is presented in [8], where the problem of minimizing the set of tuples to be locked in a relational database when handling predicate locks is studied. The presented method, however, requires a test (containment mappings) known to be NP-hard and, more importantly, does not split a locking predicate in order to grant the subset of requested tuples which are not locked: when a conflict is detected, the transaction is delayed, as in the typical approach examined in the previous section.

In order to achieve the optimal splitting strategy, the scheduler needs to know, at each moment, which entities are currently held by the active steps, and which steps are enqueued on the active steps. Clearly, a step s is enqueued on an active step t if s has been input to the scheduler after t and it requests entities denoted by t's predicate. This information can be conveniently represented by means of a binary tree, named the *splitting tree,* which is an extension of the semantic tree [3]. The *complementary pair* of a predicate p is the pair $(p, \overline{p})$, either member of which is said to be an *instance* of p.

Definition 10: Given a schedule $sch = \langle s_1, s_2, \ldots, s_m \rangle$, a *splitting tree* of sch is a labelled binary tree such that:

(i) each leaf node is either *open* or *closed,* but not both;

(ii) each non-leaf node has two outgoing links, labelled by the complementary pair of the predicate of a step in sch;

(iii) no two nodes on the path from the root to a leaf node have outgoing links labelled by the same complementary pair.

For any node n of a splitting tree,

- the *node predicate* of n, $pred(n)$, is *true* if n is the root node, otherwise it is the conjunction of the predicates found as labels in descending from the root node to n;
- the *tag* of n is "s" ("$\overline{s}$") if the incoming link of n is labelled by the positive (negative) instance of the predicate of a step s. The *signature* of n is the concatenation of the tags of the nodes encountered in descending from the root node to n;
- if n is a closed node, the *queue* of n, is a subsequence of sch in which none of the steps mentioned in the signature of n occurs. □

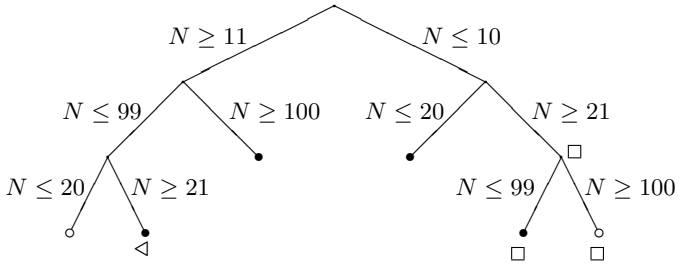

Figure 2: A splitting tree.

Example 3: Figure 2 shows a splitting tree of a schedule including the three lock steps, s_1, s_2, and s_3, such that:

$$p(s_1) = N \geq 11, \qquad p(s_2) = N \leq 20, \qquad p(s_3) = N \leq 99.$$

Closed leaf nodes are represented as disks, whereas open leaf nodes are depicted as circles. The predicate of the node indicated by the triangle is $(21 \leq N \leq 99)$, its signature is given by the string "$s_1 s_3 \overline{s_2}$", and its queue may be any subsequence of the input schedule not including s_1, s_2 and s_3. The predicate of the nodes with a square is $false$. Notice that the descendants of a node whose predicate is $false$, have $false$ as associated predicate. □

It is immediately verified that if $P_1, .., P_m$ are the predicates associated to the leaf nodes of a splitting tree and DB is any database state, then the extensions of $P_1, .., P_m$ in DB, $\varepsilon_{DB}[P_1], ..., \varepsilon_{DB}[P_m]$, are a partition of E_{DB}. In fact, in scheduling lock and unlock steps according to the optimal splitting strategy, a scheduler can be thought of as building a splitting tree whose closed leaf nodes are one-to-one with the lock steps output by the scheduler. Let us see how.

The scheduler starts with a splitting tree consisting only of the root node, whose predicate is *true,* and which is an open node, representing the fact that all the database entities are free. When the first lock step, s_1, arrives, the scheduler must output it immediately, thus granting the lock on the entities requested by s_1. This operation can be represented by attaching two nodes to the root of the splitting tree, one labelled with the predicate of s_1, $p(s_1)$, and leading to a closed node with an empty queue; the other labelled with $\overline{p(s_1)}$, and leading to an open node. Now let us assume the lock step s_2 is to be scheduled; what we want our scheduler to do is to grant the lock on the free entities among those requested by the newly arrived lock step. This means that the scheduler must output a lock step with predicate $(\overline{p(s_1)} \wedge p(s_2))$, while queueing up a step with predicate $(p(s_1) \wedge p(s_2))$, if this is satisfiable. This behavior is represented in the splitting tree by attaching two links to the currently open node: one labelled

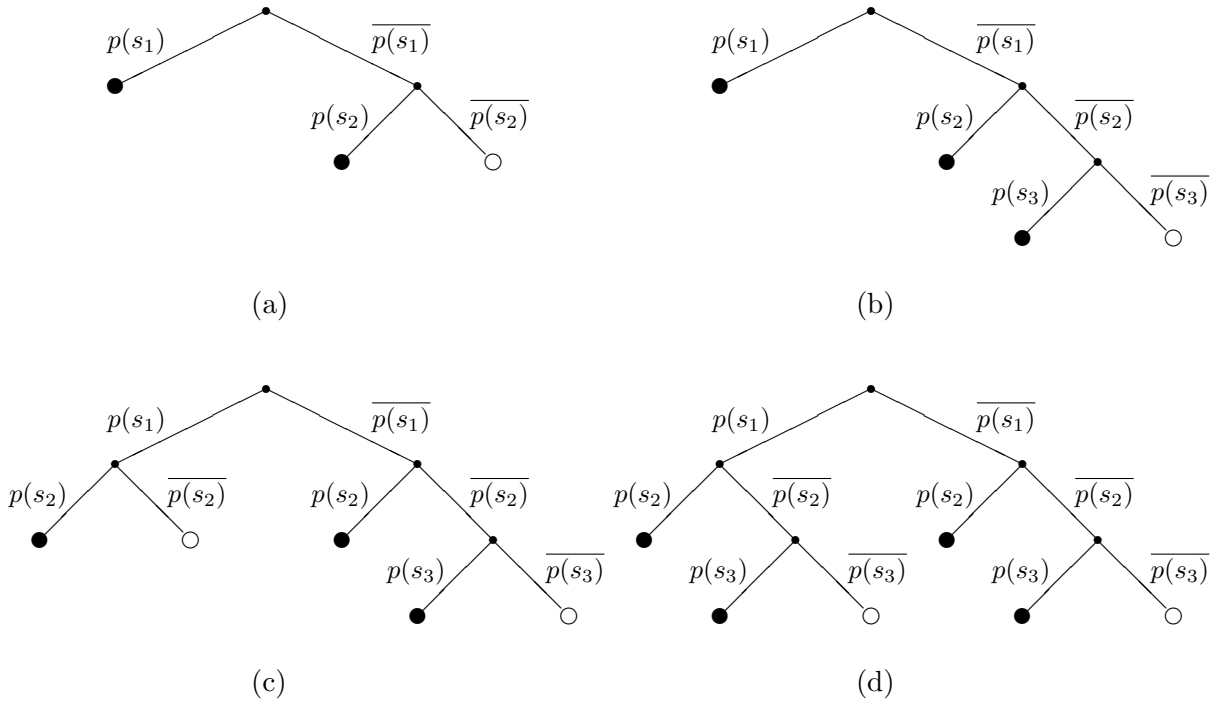

Figure 3: Successive splitting trees.

$p(s_2)$, and leading to a closed node; the other labelled $\overline{p(s_2)}$, and leading to an open node. The queue of the closed node untouched by these operations is augmented with the insertion of s_2, in case $(p(s_1) \wedge p(s_2))$ is satisfiable, that is if the newly arrived step requests some entity held by s_1. The resulting splitting tree is shown in Figure 3.a, in which the names of the predicates have been used as labels and, for a better readability, queues of pending steps are not shown. Figure 3.b presents the splitting tree after the scheduling of a third lock step s_3. In general, supposing that the current splitting tree has k open leaf nodes, $n_1, n_2, \ldots, n_k$, the scheduling of a lock step s according to the optimal splitting strategy results in the output of k lock steps $r_1, r_2, \ldots, r_k$, such that

$$p(r_i) = pred(n_i) \wedge p(s), \qquad \text{for all } 1 \leq i \leq k.$$

Correspondingly, a pair of links must be added to each open leaf node of the splitting tree, one, labelled $p(s)$, leading to a closed node; the other, labelled $\overline{p(s)}$, leading to an open node. In addition, s is inserted into the queue of each closed node c such that $(pred(c) \wedge p(s))$ is satisfiable.

An unlock step is accepted by the scheduler if its predicate is that of an active lock step, i.e. it is the predicate of a closed leaf node of the current splitting tree. Now let us suppose that the scheduler receives an unlock step

whose predicate is $p(s_1)$. This means that the entities denoted by $p(s_1)$ are now free and can be granted to the steps pending on s_1. Suppose both s_2 and s_3 are in this state and that the scheduler considers them in their arrival order. In this case, we want the scheduler to output two lock steps; one with predicate $(p(s_1) \wedge p(s_2))$, which would leave free the entities denoted by the predicate $(p(s_1) \wedge \overline{p(s_2)})$; the other with predicate $(p(s_1) \wedge \overline{p(s_2)} \wedge p(s_3))$. The entities denoted by $(p(s_1) \wedge \overline{p(s_2)} \wedge \overline{p(s_3)})$ are now free and ready to be allocated to forthcoming steps. The modifications to the splitting tree needed to represent these operations are reported in Figures 3.c and 3.d. The queue of the closed node corresponding to the first output step includes s_3, whereas that associated to the node of the second step is empty. In general, the scheduling of an unlock step can be seen as the composition of the scheduling of several lock steps, one for each step pending on the unlocked node. If we let $\langle q_1, q_2, \ldots, q_k \rangle$ be any permutation of the predicates of the pending steps, in order to follow the optimal splitting strategy the scheduler must output k lock steps $r_1, r_2, \ldots, r_k$, such that

$$p(r_j) = p(s) \wedge q_j \wedge \overline{q_1} \wedge \overline{q_2} \wedge \ldots \overline{q_{j-1}}, \qquad \text{for all } 1 \leq j \leq k.$$

The entities left free after this re-scheduling are those denoted by the predicate:

$$p(s) \wedge \overline{q_1} \wedge \overline{q_2} \wedge \ldots \wedge \overline{q_k}.$$

The corresponding operations on the tree are the obvious generalization of those presented in Figures 3.c and 3.d.

The question naturally arises whether the order in which pending steps are re-scheduled impacts on the performance of the scheduler. The answer to this question is positive: different re-scheduling orders may make a difference on how efficient the scheduler is in re-assigning database entities to pending steps. However, this difference depends on the current database state, and taking it into account would go against the philosophy of predicate locking, which solves the phantom problem just because it operates at the level of predicates and not on the current database state. For this reason, in the rest of the paper we will adopt the simplest re-scheduling policy: pending steps will be selected for re-scheduling according to their arrival order.

Another question concerns the size of the splitting tree. Let us consider again the tree shown in Figure 3.d, and suppose that an unlock step with predicate $(\overline{p(s_1)} \wedge \overline{p(s_2)} \wedge p(s_3))$ is input to the scheduler. The resulting tree is shown in Figure 4.a, and an equivalent, simplified version of this tree is given in Figure 4.b. The simplification consists in the collapse of two sibling open leaf nodes into their parent node, and is justified by the intuitive criterion that

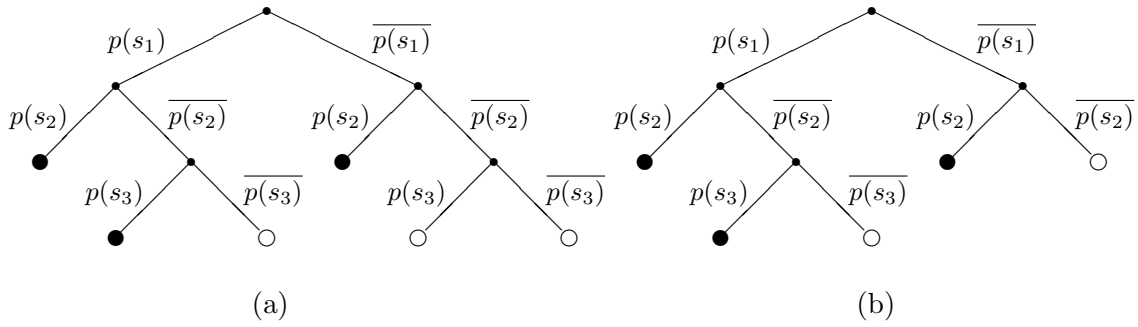

Figure 4: A splitting tree simplification.

the two trees denote, in each database state, the same sets of free and locked database entities. It is easily verified that, by applying this simplification, the scheduling of the unlock steps corresponding to all the closed leaf nodes will eventually produce the initial splitting tree, consisting of just the root node.

4.2 The scheduler

We are now in the position of formalizing the behavior of our optimal scheduler, which will be done by means of optimal tree states. An optimal tree state is the formal representation of a splitting tree generated by the optimal splitting strategy.

Definition 11: A *tree state* is a 4-tuple $\langle \Lambda, \Phi, q, z \rangle$, where: Λ and Φ are sets of predicates, q is a total function from Λ to schedules, and z is a total function from $(\Lambda \cup \Phi)$ to signatures. The *state predicate of* a tree state is the disjunction of the predicates in Λ. □

As will be clear in a moment, a tree state is intended to model a splitting tree, hence its four components will stand, respectively, for closed leaf node predicates, open leaf node predicates, queues of the closed leaf nodes, and signatures of the leaf nodes.

In order to formally capture the tree simplification discussed in the previous section, we next define a function that applies this simplification to tree states ("'ξ" stands for the character string "ξ").

Definition 12: Given a tree state $\tau = \langle \Lambda, \Phi, q, z \rangle$, two predicates ϕ_1 and ϕ_2 in Φ are said to be *resolving* if and only if $z(\phi_1) = `\xi s$ and $z(\phi_2) = `\xi \overline{s}$; the *resolvent* of ϕ_1 and ϕ_2 is the predicate $(\phi_1 \vee \phi_2)$, whose signature is given by $`\xi$. The *simplification* of τ, $\varsigma(\tau)$, is the state $\tau = \langle \Lambda, \Phi', q, z' \rangle$, where Φ' is obtained

Table II: A tree state with resolving predicates

Λ	Φ	q	z
$p(s_1) \wedge p(s_2)$		$\langle s_3 \rangle$	$`s_1 s_2$
$p(s_1) \wedge \overline{p(s_2)} \wedge p(s_3)$		$\Box$	$`s_1 \overline{s_2} s_3$
$\overline{p(s_1)} \wedge p(s_2)$		$\langle s_3 \rangle$	$`\overline{s_1} s_2$
	$p(s_1) \wedge \overline{p(s_2)} \wedge \overline{p(s_3)}$		$`s_1 \overline{s_2 s_3}$
	$\overline{p(s_1)} \wedge \overline{p(s_2)} \wedge p(s_3)$ $\star$		$`\overline{s_1 s_2} s_3$
	$\overline{p(s_1)} \wedge \overline{p(s_2)} \wedge \overline{p(s_3)}$ $\star$		$`\overline{s_1 s_2 s_3}$

by recursively replacing in Φ each pair of resolving predicates by their resolvent, and z' is obtained by modifying z accordingly. $\Box$

The last definition takes into account the fact that more than one simplification may take place upon scheduling an unlock step. The recursion of the simplification process is clearly harmless. First, as shown in Section 4.1, at most one open leaf node is added to the splitting tree upon scheduling an unlock step, hence there will always be at most one pair of resolving predicates. Second, the simplification decreases the size of Φ and that of signatures, hence the process will always terminates in a linear number of steps.

Among the infinitely many tree states introduced by Definition 11, we wish to select those corresponding to an optimal splitting strategy, as illustrated by the following example.

Example 4: The tree state corresponding to the splitting tree of Figure 4.a, is given in Table II. The predicates marked with a star are resolving. In the simplified state (not shown), these are replaced by their resolvent, given by $(\overline{p(s_1)} \wedge \overline{p(s_2)})$, whose signature is given by "$\overline{s_1 s_2}$". It is quickly verifiable that the simplified state corresponds to the splitting tree of Figure 4.b. $\Box$

In the following definition, if s is a sequence, x a variable and P a condition on s, we will let the expression "$s|x : P$" stand for the subsequence of s consisting of those elements x satisfying P. The function *cat* concatenates its string arguments.

Definition 13: Let *sch* be the schedule $s_1 s_2 \ldots s_m$. The *optimal scheduling of sch* is the sequence of tree states $\langle \tau_1, \ldots, \tau_m \rangle$, where $\tau_k = \langle \Lambda_k, \Phi_k, q_k, z_k \rangle$ is the *k-th optimal tree state on sch,* $1 \leq k \leq m$, inductively defined as follows:

$\underline{\tau_1}$:

$$\Lambda_1 \quad = \quad \{p(s_1)\}$$

$$
\begin{aligned}
\Phi_1 &= \{\overline{p(s_1)}\} \\
q_1(\lambda) &= \Box \\
z_1(\theta) &= \begin{cases} `s_1 & \text{if } \theta \text{ is } p(s_1) \\ `\overline{s_1} & \text{if } \theta \text{ is } \overline{p(s_1)} \end{cases}
\end{aligned}
$$

for all $\lambda \in \Lambda_1$ and $\theta \in (\Lambda_1 \cup \Phi_1)$.

$\underline{\tau_{k+1}}$, if s_{k+1} is a lock step:

$$
\begin{aligned}
\Lambda_{k+1} &= \Lambda_k \cup \{\phi \wedge p(s_{k+1}) |\ \phi \in \Phi_k \text{ and } \phi \wedge p(s_{k+1}) \text{ is satisfiable}\} \\
\Phi_{k+1} &= \{\phi \wedge \overline{p(s_{k+1})} |\ \phi \in \Phi_k \text{ and } \phi \wedge \overline{p(s_{k+1})} \text{ is satisfiable}\} \\
q_{k+1}(\lambda) &= \begin{cases} q_k(\lambda) \sqcup s_{k+1} & \text{if } \lambda \in \Lambda_k \text{ and } (\lambda \wedge p(s_{k+1})) \text{ is satisfiable} \\ q_k(\lambda) & \text{if } \lambda \in \Lambda_k \text{ and } (\lambda \wedge p(s_{k+1})) \text{ is unsatisfiable} \\ \Box & \text{if } \lambda \notin \Lambda_k \end{cases} \\
z_{k+1}(\theta) &= \begin{cases} cat(z_k(\phi), `s_{k+1}) & \text{if } \theta \text{ is } \phi \wedge p(s_{k+1}) \\ cat(z_k(\phi), `\overline{s_{k+1}}) & \text{if } \theta \text{ is } \phi \wedge \overline{p(s_{k+1})} \\ z_k(\theta) & \text{otherwise} \end{cases}
\end{aligned}
$$

for all $\lambda \in \Lambda_{k+1}$ and $\theta \in (\Lambda_{k+1} \cup \Phi_{k+1})$.

$\underline{\tau_{k+1}}$, if s_{k+1} is an unlock step, is given by the simplification of the tree state τ_u, defined as follows. Let $q_k(p(s_{k+1}))$ be the sequence $\langle s_{i_1} s_{i_2} \ldots s_{i_p} \rangle$; then:

$$
\begin{aligned}
\Lambda_u &= \Lambda_k - \{p(s_{k+1})\} \cup \{\alpha_j |\ 1 \le j \le p \text{ and } \alpha_j \text{ is satisfiable}\} \quad \text{where} \\
\alpha_j &= p(s_{k+1}) \wedge p(s_{i_j}) \wedge \bigwedge_{l=1}^{j-1} \overline{p(s_{i_l})} \\
\Phi_u &= \begin{cases} \Phi_k \cup \{\beta\} & \text{if } \beta \text{ is satisfiable} \\ \Phi_k & \text{otherwise} \end{cases} \quad \text{where} \\
\beta &= p(s_{k+1}) \wedge \bigwedge_{j=1}^{p} \overline{p(s_{i_j})} \\
q_u(\lambda) &= \begin{cases} q_k(\lambda) & \text{if } \lambda \in \Lambda_k \\ s_{i_{j+1}} \ldots s_{i_p} | s : (\alpha_j \wedge p(s)) \text{ is satisfiable} & \text{if } \lambda = \alpha_j \end{cases} \\
z_u(\theta) &= \begin{cases} cat(z_k(p(s_{k+1})), `\overline{s_{i_1}}, \ldots, `\overline{s_{i_{j-1}}}, `s_{i_j}) & \text{if } \theta \text{ is } \alpha_j \\ cat(z_k(p(s_{k+1})), `\overline{s_{i_1}}, \ldots, `\overline{s_{i_p}}) & \text{if } \theta \text{ is } \beta \\ z_k(\theta) & \text{otherwise} \end{cases}
\end{aligned}
$$

for all $\lambda \in \Lambda_u$ and $\theta \in (\Lambda_u \cup \Phi_u)$. □

The pseudo-Pascal procedure *tree_sch,* given in Figure 5, implements in a straightforward way the optimal splitting strategy, by operating on the current optimal tree state as defined above.

```
procedure tree_sch (s: step)
l: step;   λ, φ, φ': predicate
begin
if a(s) = 'lock' then
  begin
  for each λ in Λ if satisfiable(λ ∧ p(s)) then append(q(λ), s)
  for each φ in Φ if satisfiable(φ ∧ p(s)) then begin
    Φ ← Φ − {φ};   λ ← φ ∧ p(s);   q(λ) ← □;   z(λ) ← cat(z(φ), 's);   Λ ← Λ ∪ {λ}
    output − lock − step(λ)
    if satisfiable(φ ∧ ¬p(s)) then begin
      φ' ← φ ∧ ¬p(s);   z(φ') ← cat(z(φ), '¬s);   Φ ← Φ ∪ {φ'}
      end
    end
  end
else if a(s) = 'unlock' then begin
  output(s)
  φ ← p(s);   Λ ← Λ − {φ}
  if empty(q(φ)) then simplify(Φ ∪ {φ})
  else for each l in q(φ) if satisfiable(φ ∧ p(l)) then begin
    λ ← φ ∧ p(l);   q(λ) ← cut(q(φ), λ);   z(λ) ← cat(z(φ), 'l);   Λ ← Λ ∪ {λ}
    output − lock − step(λ)
    φ ← φ ∧ ¬p(l);   z(φ) ← cat(z(φ), '¬l)
    end
  if satisfiable(φ) then Φ ← Φ ∪{φ}
  end
end
```

Figure 5: The optimal predicate locking scheduler

In scheduling a newly arrived lock step s, *tree_sch* tests if s's predicate conflicts with the predicate (λ) of each closed leaf node; for each such detected conflict, *tree_sch* appends s to λ's queue, waiting to be scheduled. Successively, *tree_sch* test whether s's predicate is disjoint from the predicate (ϕ) of each open leaf node; if not, ϕ is removed from Φ, because some of the entities it denotes are going to be granted to s, as required by the optimal splitting strategy. The predicate denoting these entities (given by $\phi \wedge p(s)$) is computed, it is given the empty queue and the proper signature, and it is added to Λ. The corresponding lock step is output. The variable λ is used to hold this predicate during the operations just described. If some entity denoted by ϕ is left free, $\phi \wedge \overline{p(s)}$ is satisfiable and is thus added to Φ after setting its signature. Notice that if for some $\phi \in \Phi$, $\phi \wedge p(s)$ is not satisfiable, *tree_sch* leaves it unchanged; and this is exactly what the specification of τ prescribes, since in this case $\phi \wedge \overline{p(s)}$ is equivalent to ϕ.

When an unlock step s is received, *tree_sch* outputs it, and if no step is pending on the unlocked predicate, we have the kind of situation presented in Figure 4.a. The simplification discussed in the previous section may thus take place, under the responsibility of the procedure *simplify,* not presented as not particularly interesting. Otherwise, *tree_sch* enters a loop in which it re-schedules the pending steps, in the same order as they appear in the involved queue; to this end, the predicate of each such step l is matched with the predicate denoting the currently free entities (ϕ), to ascertain whether l references some of the freed entities. If not, l is no longer considered. If yes, a new predicate λ is generated whose queue is set to the subsequence of $q(\phi)$ following l and consisting of the steps conflicting with it (this is done by the *cut* procedure, which is not presented). The signature of λ is set to the proper value and λ is finally inserted into Λ. The predicate denoting the free entities, ϕ, is modified in order to reflect the lock just granted, and so is its signature. If, at the end of the re-scheduling loop, ϕ is satisfiable, then some of the unlocked entities have not been re-assigned to a pending step; to keep track of this, ϕ is added to Φ.

In order to prove the correctness of *tree_sch,* we need the following lemma.

Lemma 1: For any schedule *sch* in the domain of *tree_sch,* database state and time point t,

$$C(tree\_sch(sch), e, t) = \begin{cases} 1 & \text{if } e \text{ satisfies a predicate in } \Lambda \text{ at } t \\ 0 & \text{otherwise.} \end{cases}$$

Proof: A lock step is output if and only if a closed leaf node is created, hence

$$C(tree\_sch(sch), e, t) = 0$$

if e is not referenced by the predicate of a closed leaf node. In addition, given any two closed leaf node predicates p and q, there exists an input predicate c such that c is used to compute p and $\overline{c}$ is used to compute q, or vice versa. Hence no entity e can satisfy two closed leaf node predicates, hence

$$C(tree\_sch(sch), e, t) = 1$$

for all entities satisfying a closed leaf node predicate. □

Proposition 3: *tree_sch* is correct.

Proof: We must show that conditions (i) to (iii) of Definition 8 hold. The proof for condition (ii) is trivial, whereas (iii) directly follows from the previous lemma. We will then give the proof of condition (i), by specifying the required bijective mapping. For a generic database state DB, let s_k be the k-th input lock step of a schedule sch in the domain of *tree_sch,* and $e \in E_{DB}$ a database entity satisfying $p(s_k)$ in DB, so that $(s_k, e) \in LS(sch)$. There is exactly one leaf node n, with predicate P, such that e satisfies P. If n is an open leaf node, upon scheduling s_k the scheduler outputs a lock step s whose predicate is $P \wedge p(s_k)$, satisfied by e, so that $(s, e) \in LS(tree\_sch(sch))$. As $time(s_k) = time(s)$, we can pose $(s_k, e)R(s, e)$. If n is a closed leaf node, s_k is queued on n, and, eventually, when it will be dequeued to be scheduled, the scheduler will be in the same condition as in the previous case, and will output a lock step s', such that $time(s_k) < time(s')$. We can then set $(s_k, e)R(s', e)$. The mapping R so built is clearly total and injective from $LS(sch)$ to $LS(tree\_sch(sch))$. Assume that it is not surjective, that is that there exists a lock step s_o output by the scheduler and an entity e referenced by s_o in DB, i.e. $(s_o, e) \in LS(tree\_sch(sch))$, such that there is no pair in R whose second member is (s_o, e). There are two possibilities: (1) s_o is output upon the scheduling of a lock step s_a; in this case, e satisfies the predicate $pred(o) \wedge p(s_a)$, where o is an open leaf node of the current tree; but then e satisfies $p(s_a)$, hence $(s_a, e) \in LS(sch)$, and, by construction, $(s_a, e)R(s_o, e)$, so we have a contradiction. (2) s_o is output upon the scheduling of an unlock step s_b whose predicate is that of a closed node c. In this case, e satisfies the predicate

$$pred(c) \wedge \overline{p(s_d)} \wedge \overline{p(s_{d+1})} \wedge \ldots \wedge p(s_{d+h})$$

where $d \geq 0$, $h \geq 1$, and s_{d+h} is the last input lock step received by the scheduler. But then e satisfies $p(s_{d+h})$, hence $(s_{d+h}, e) \in LS(sch)$, and, by construction, $(s_{d+h}, e)R(s_o, e)$. Thus in this case too we have a contradiction. Therefore R is the total, injective and surjective mapping from $LS(sch)$ to $LS(tree\_sch(sch))$ required by Definition 8. □

4.3 Performance

According to [11], the performance of the scheduler is given by the *concurrency* and the *efficiency* of the scheduler. Concurrency measures the degree of parallelism allowed by the scheduler, and is directly related to the amount of data that the scheduler is able to grant without compromising the correctness of execution of transactions. As the optimal splitting strategy maximizes such amount, the concurrency of *tree_sch* is maximal.

The efficiency of a scheduler is a measure of the complexity of the algorithm implementing the scheduler. A scheduler is efficient if it fulfills the following two conditions [11]: (a) the size of the state data structure is at any moment bounded by a polynomial in the size of the initial state and the number of steps that have arrived so far, and (b) the number of steps needed to test whether a step will be output or will join the queue, and to update the state to reflect the information obtained from the last step, are both polynomial in the size of the state.

The state data structure used by *tree_sch* is a tree state. Since both the size of queues and that of signatures are no greater than the number of steps received so far, we can focus on the two other components of the state. In particular, to prove that *tree_sch* satisfies condition (a), we must show: that the number of leaf nodes of the current splitting tree is bounded by a polynomial in the number of steps that have arrived so far, and that so is the size of the predicates associated to these nodes. As far as the former issue is concerned, the following proposition settles the case.

Proposition 4: For any schedule sch,

$$\begin{aligned} |\Phi_k| &\leq k, \\ |\Lambda_k| &\leq \frac{1}{2}(k^2 - k) + 1, \qquad \text{for all } 1 \leq k \leq |sch|. \end{aligned}$$

Proof: Let us first consider the size of the set of open leaf node predicates. Considering the worst case, when all the predicates generated by the optimal splitting strategy are satisfiable and no simplification takes place, we have:

$$|\Phi_1| = 1$$
$$|\Phi_{k+1}| = \begin{cases} |\Phi_k| & \text{if } s_{k+1} \text{ is a lock step} \\ |\Phi_k| + 1 & \text{if } s_{k+1} \text{ is an unlock step} \end{cases}$$

Notice that in case of unlock, the predicate β of Definition 13 is added to Φ. The part of the proposition for Φ_k follows immediately. Now let us consider the closed leaf node predicates. In case of a lock step, the worst case is when all predicates generated by the optimal splitting strategy are satisfiable; in case of

an unlock step, the worst case is when no simplification takes place, the steps to be re-scheduled are all the lock steps received so far except the first (which is never enqueued), and all these steps generate a satisfiable predicate α_j. We have therefore:

$$|\Lambda_1| = 1$$
$$|\Lambda_{k+1}| \leq \begin{cases} |\Lambda_k| + |\Phi_k| & \text{if } s_{k+1} \text{ is a lock step} \\ |\Lambda_k| + k - 2 & \text{if } s_{k+1} \text{ is an unlock step.} \end{cases}$$

As we have already proved that $|\Phi_k| \leq k$, we have

$$|\Lambda_{k+1}| \leq |\Lambda_k| + k.$$

By a simple induction argument, the proposition follows. □

The size of the predicates associated to the leaf nodes of the tree depends on the form of the predicates of the input lock steps, as does the complexity of the satisfiability test involved in condition (b) above. Without loss of generality, we assume these predicates to be of the form:

$$\bigwedge_{i=1}^{pos} c_i \wedge \bigwedge_{j=1}^{neg} \overline{c}_{pos+j}$$

where each c_k is the predicate of an input lock step (not necessarily step k), and either *pos* or *neg* may be 0, but not both. As already pointed out, this test is not effective in the general case of first-order languages, and there is no evidence that the restricted first-order language that we have assumed as the database predicate language makes things any easier.

There is a vast literature on methods for testing logical properties of database predicates such as equivalence, implication (sometimes called containment), and satisfiability (sometimes called disjointness). However, what is generally offered by these studies is a decision procedure, whereas our scheduler requires a procedure that, beside testing satisfiability, also computes complex predicates. In particular, [9] studies the implication and equivalence problems for conjunctive queries, and does not deal with satisfiability, whereas [4] provides an algorithm for deciding the disjointness of conjunctive queries. This procedure can be used as a basis for an instance of the scheduler *typ_sch*, as the author envisages a predicate locking scheduler based on a satisfiability check. [7] shows that testing the satisfiability of quantifier-free formulae in Conjunctive Normal Form is NP-hard. However, [12] presents a polynomial time algorithm that tests the satisfiability of the conjunction of simple and complex atomic formulae containing no negated equality formulae. [2] proposes to take such formulae as the disjuncts of a DNF formula, so that by a polynomially bounded number

of applications of the above mentioned algorithm, the satisfiability of the DNF can be tested. However, for the reason pointed out above, we will assume as predicates of the input lock steps the simple formulae of [12], obtaining the language given in the next definition.

Definition 14: Given a database $\langle E, \mathcal{D}, \mathcal{S} \rangle$, the *simplified* database predicate language $\mathcal{LS}$, is the subset of $\mathcal{L}$ defined as follows:

(i) the *sorts* and the *alphabet* of $\mathcal{LS}$ are the same as those of $\mathcal{L}$, except for the predicate symbols, of which there are two $\prec_i$ and $\succ_i$ for each sort, beside the equality symbol;

(ii) the *terms* of $\mathcal{LS}$ are the constant and the atomic function terms of $\mathcal{L}$;

(iii) the *atomic formulae,* or simply the *atoms,* of $\mathcal{LS}$ are the simple atomic formulae of $\mathcal{L}$, built with constant and atomic function terms, and the always true atom *true;*

(iv) the *well-formed formulae* of $\mathcal{LS}$ are built out of the atoms of $\mathcal{LS}$ by using the logical connective $\wedge$ in the standard way. □

The semantics of $\mathcal{LS}$ is the standard first-order semantics, where each $\succ_i$ is to be interpreted as the negation of the corresponding $\prec_i$, that is:

$$\succ_i (\phi, \delta) \equiv \overline{\prec_i (\phi, \delta)}$$

where ϕ is an atomic function term and δ is a constant term. This introduces a restricted form of negation. In order to express in a compact way the conjunction of mutually consistent $\prec_i$ and $\succ_i$ atoms, we further introduce m ternary predicate symbols $\ll_i$, that is, if $a \leq b$,

$$\prec_i (\phi, b) \wedge \succ_i (\phi, a) \equiv \ll_i (\phi, a, b).$$

As we have assumed a database consisting of one domain, the natural numbers ordered by the $\leq$ relation, the simplified language we will deal with has four predicate symbols: the binary symbols $=$, $\leq$ and $>$, and the ternary symbol $\leq\leq$. The binary symbols $<$ and $\geq$ can then be added by defining them in the proper way.

A formula of $\mathcal{LS}$ is a conjunction of simple atoms, in which the same function term may appear an arbitrary number of times. It can be shown that all the atoms on the same term can be reduced to a single atom, so that we may assume that a formula contains exactly one atom per function symbol, as the missing atoms can be replaced by *true* without altering the semantics of the formula. Analogously, it can be shown that the negation of an $\mathcal{LS}$ formula is not in $\mathcal{LS}$,

as the negation of an interval atomic predicate yields the disjunction of two atoms.

Proposition 5: The predicate $\bigwedge_{i=1}^{pos} c_i \wedge \bigwedge_{j=1}^{neg} \overline{c}_{pos+j}$, where each c_k, $1 \leq k \leq pos + neg$, is a formula, in the worst case is equivalent to the disjunction of $(2 \cdot n)^{neg}$ formulae, where n is the number of the properties of the database.
Proof: In the worst case, each $\overline{c}_{pos+j}$ yields the disjunction of a pair of atoms for each function symbol, that is the disjunction of $2 \cdot n$ atoms. The conjunction of neg of these disjunctions is equivalent, by distributivity, to a disjunction of $(2 \cdot n)^{neg}$ formulae. □

5 An efficient optimal scheduler

The last proposition tells us that computing leaf node predicates by means of the distributive and De Morgan laws has an exponential cost. However, the problem is not inherently intractable, and in this section we will present a method to efficiently solve it.

5.1 The grid method

Let us suppose that the schedule $s_1 s_2 \ldots$ is input to *tree_sch,* where s_1 and s_2 are lock steps such that:

$$\begin{aligned} p(s_1) &= (10 \leq N_1 \leq 30) \wedge (N_2 \geq 16) \\ p(s_2) &= (N_1 \geq 20) \wedge (10 \leq N_2 \leq 20). \end{aligned}$$

As we have seen, *tree_sch* responds to the arrival of s_1 by outputting it; upon receiving s_2, it must output a lock step whose predicate is $(\overline{p(s_1)} \wedge p(s_2))$, given by:

$$[(N_1 \geq 31) \wedge (10 \leq N_2 \leq 20)] \vee [(N_1 \geq 20) \wedge (10 \leq N_2 \leq 15)]. \quad (1)$$

In addition, *tree_sch* must compute the predicate $(\overline{p(s_1)} \wedge \overline{p(s_2)})$, which turns out to be:

$$(N_1 \leq 9) \vee [(N_1 \geq 31) \wedge (N_2 \geq 21)] \vee [(N_1 \leq 19) \wedge (N_2 \leq 15)] \vee (N_2 \leq 9). \quad (2)$$

Now, let us consider a geometrical representation of the database space, where each point of a 2-dimensional space is taken to represent a combination of the values of the database properties. A point of this space can be associated to the set of the database entities having the coordinates of the point as property values in the current database state; moreover, since the interpretation of the constant and predicate symbols of the database language is state independent,

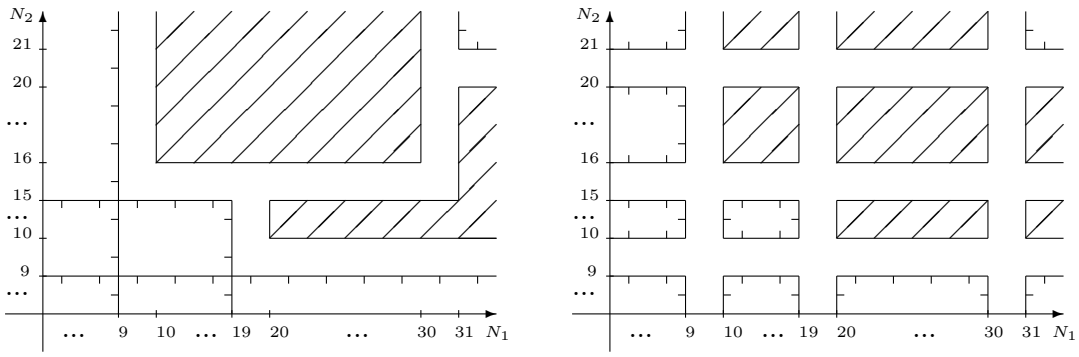

Figure 6: Geometrical interpretations of a scheduler state.

we can associate to a formula of $\mathcal{LS}$ a convex region of the space. According to this representation, the state of the scheduler in the above example can be depicted as in Figure 6.a, where the shadowed regions contain the points associated to the locked entities and the remaining areas the points associated to the free entities. As Figure 6.a shows, the formulae (1) and (2) are semantically redundant; in particular, the region $(N_1 \leq 9) \wedge (N_2 \leq 9)$ is denoted by three disjuncts of (2) (namely the first, third and fourth disjunct), and the region $(N_1 \geq 31) \wedge (10 \leq N_2 \leq 15)$ is contained in the denotation of both the disjuncts constituting (1). This redundancy can be eliminated by representing locked and free database entities via the predicates corresponding to the regions in the partition presented in Figure 6.b. This partition is the product of partitions of the property domains, and can be conveniently represented and manipulated by means of an extension of the *grid directory* [10], hence the name of *grid method.*

A grid directory for k attributes consists of two parts: first, a dynamic k-dimensional array, called the *grid array;* second, k 1-dimensional arrays called *linear scales,* each defining a partition in intervals of the domain of a property. In a grid structure, each grid array cell contains a pointer to a sequence of records. In our structure, which we call the *grid state,* the cells of the grid array will correspond in a many-to-one way to the leaf nodes of the current splitting tree; thus, each cell will contain the information associated to its corresponding node, that is:

- a binary value, G, indicating the leaf node type; we will use $G = 1$ for closed nodes, and $G = 0$ for open nodes;
- the queue Q of the steps pending on the node;
- the signature Z of the node.

The grid state representing the situation of the example is given in the following table, where, following the convention adopted for the geometrical representa-

tion, the domain of N_1 is placed horizontally, whereas the domain of N_2 is placed vertically. Each cell of the grid array gives, in order, the G-, Q- and Z-value on the corresponding region of the partition.

	(9]	[10,19]	[20,30]	[31)
[21)	0 □ $\overline{s_1 s_2}$	1 □ s_1	1 □ s_1	0 □ $\overline{s_1 s_2}$
[16,20]	0 □ $\overline{s_1 s_2}$	1 □ s_1	1 $\langle s_2 \rangle$ s_1	1 □ $\overline{s_1} s_2$
[10,15]	0 □ $\overline{s_1 s_2}$	0 □ $\overline{s_1 s_2}$	1 □ $\overline{s_1} s_2$	1 □ $\overline{s_1} s_2$
(9]	0 □ $\overline{s_1 s_2}$	0 □ $\overline{s_1 s_2}$	0 □ $\overline{s_1 s_2}$	0 □ $\overline{s_1 s_2}$

Let us assume that the scheduler now receives the step:

$$s_3 : \ lock[(15 \leq N_1 \leq 25) \wedge (N_2 \leq 12)]$$

In order to represent the predicate of this step in the above grid state, three intervals must be split, namely:

- the interval $[10, 15]$ in N_2's domain, into the intervals $[10, 12]$ and $[13, 15]$;
- the interval $[10, 19]$ in N_1's domain, into the intervals $[10, 14]$ and $[15, 19]$;
- the interval $[20, 30]$ in N_1's domain, into the intervals $[20, 25]$ and $[26, 30]$.

The resulting grid state is:

	(9]	[10,14]	[15,19]	[20,25]	[26,30]	[31)
[21)	0 □ $\overline{s_1 s_2 s_3}$	1 □ s_1	1 □ s_1	1 □ s_1	1 □ s_1	0 □ $\overline{s_1 s_2 s_3}$
[16,20]	0 □ $\overline{s_1 s_2 s_3}$	1 □ s_1	1 □ s_1	1 $\langle s_2 \rangle$ s_1	1 $\langle s_2 \rangle$ s_1	1 □ $\overline{s_1} s_2$
[13,15]	0 □ $\overline{s_1 s_2 s_3}$	0 □ $\overline{s_1 s_2 s_3}$	0 □ $\overline{s_1 s_2 s_3}$	1 □ $\overline{s_1} s_2$	1 □ $\overline{s_1} s_2$	1 □ $\overline{s_1} s_2$
[10,12]	0 □ $\overline{s_1 s_2 s_3}$	0 □ $\overline{s_1 s_2 s_3}$	1 □ $\overline{s_1 s_2} s_3$	1 $\langle s_3 \rangle$ $\overline{s_1} s_2$	1 □ $\overline{s_1} s_2$	1 □ $\overline{s_1} s_2$
(9]	0 □ $\overline{s_1 s_2 s_3}$	0 □ $\overline{s_1 s_2 s_3}$	1 □ $\overline{s_1 s_2} s_3$	1 □ $\overline{s_1 s_2} s_3$	0 □ $\overline{s_1 s_2 s_3}$	0 □ $\overline{s_1 s_2 s_3}$

The set of the database space points denoted by the predicate of the newly arrived step is represented by a subset of the grid cells. The step is enqueued on those cells from this subset which are contained in a cell of the previous state with a 1 G-value. For the other cells, a lock step is output and the G-value of these cells is set to 1. The Z-value of the new state's cells is also properly set.

Upon unlocking, the cells of the grid array that are denoted by the unlock predicate are identified; if their queue is empty, then no step is waiting for the unlocked entities, so their G-value is turned to 0; otherwise, the first step is popped from the queue and the G-value is left at 1, meaning that the dequeued step is granted the corresponding entities. For instance, suppose that the step:

$$s_4 : \ unlock[((N_1 \geq 20) \wedge (10 \leq N_2 \leq 15)) \vee ((N_1 \geq 31) \wedge (10 \leq N_2 \leq 20))]$$

unlocking the entities denoted by the predicate $\overline{p(s_1)} \wedge p(s_2)$, is now input to the scheduler. Notice that the step's predicate is not part of the database language $\mathcal{LS}$, which only applies to input lock steps. The resulting grid state is:

	(9]			[10,14]			[15,19]			[20,25]			[26,30]			[31)		
[21)	0	□	$\overline{s_1 s_2 s_3}$	1	□	s_1	1	□	s_1	1	□	s_1	1	□	s_1	0	□	$\overline{s_1 s_2 s_3}$
[16,20]	0	□	$\overline{s_1 s_2 s_3}$	1	□	s_1	1	□	s_1	1	$\langle s_2 \rangle$	s_1	1	$\langle s_2 \rangle$	s_1	0	□	$\overline{s_1 s_2 s_3}$
[13,15]	0	□	$\overline{s_1 s_2 s_3}$	0	□	$\overline{s_1 s_2 s_3}$	0	□	$\overline{s_1 s_2 s_3}$	0	□	$\overline{s_1} s_2 \overline{s_3}$	0	□	$\overline{s_1} s_2 \overline{s_3}$	0	□	$\overline{s_1 s_2 s_3}$
[10,12]	0	□	$\overline{s_1 s_2 s_3}$	0	□	$\overline{s_1 s_2 s_3}$	1	□	$\overline{s_1 s_2} s_3$	1	□	$\overline{s_1} s_2 s_3$	0	□	$\overline{s_1} s_2 \overline{s_3}$	0	□	$\overline{s_1 s_2 s_3}$
(9]	0	□	$\overline{s_1 s_2 s_3}$	0	□	$\overline{s_1 s_2 s_3}$	1	□	$\overline{s_1 s_2} s_3$	1	□	$\overline{s_1 s_2} s_3$	0	□	$\overline{s_1 s_2 s_3}$	0	□	$\overline{s_1 s_2 s_3}$

By looking at the signatures of the regions associated to free entities, we can see that no simplification of the corresponding splitting tree takes place. But upon receiving the step:

$$s_5 : \ \mathit{unlock}[((15 \leq N_1 \leq 25) \wedge (N_2 \leq 9)) \vee ((15 \leq N_1 \leq 19) \wedge (10 \leq N_2 \leq 12))],$$

unlocking the entities denoted by $\overline{p(s_1)} \wedge \overline{p(s_2)} \wedge p(s_3)$, the following grid is produced:

	(9]			[10,14]			[15,19]			[20,25]			[26,30]			[31)		
[21)	0	□	$\overline{s_1 s_2 s_3}$	1	□	s_1	1	□	s_1	1	□	s_1	1	□	s_1	0	□	$\overline{s_1 s_2 s_3}$
[16,20]	0	□	$\overline{s_1 s_2 s_3}$	1	□	s_1	1	□	s_1	1	$\langle s_2 \rangle$	s_1	1	$\langle s_2 \rangle$	s_1	0	□	$\overline{s_1 s_2 s_3}$
[13,15]	0	□	$\overline{s_1 s_2 s_3}$	0	□	$\overline{s_1 s_2 s_3}$	0	□	$\overline{s_1 s_2 s_3}$	0	□	$\overline{s_1} s_2 \overline{s_3}$	0	□	$\overline{s_1} s_2 \overline{s_3}$	0	□	$\overline{s_1 s_2 s_3}$
[10,12]	0	□	$\overline{s_1 s_2 s_3}$	0	□	$\overline{s_1 s_2 s_3}$	0	□	$\overline{s_1 s_2} s_3$	1	□	$\overline{s_1} s_2 s_3$	0	□	$\overline{s_1} s_2 \overline{s_3}$	0	□	$\overline{s_1 s_2 s_3}$
(9]	0	□	$\overline{s_1 s_2 s_3}$	0	□	$\overline{s_1 s_2 s_3}$	0	□	$\overline{s_1 s_2} s_3$	0	□	$\overline{s_1 s_2} s_3$	0	□	$\overline{s_1 s_2 s_3}$	0	□	$\overline{s_1 s_2 s_3}$

in which there are resolving predicates, namely those associated to the cells denoting free entities and whose signatures are: "$\overline{s_1 s_2 s_3}$" and "$\overline{s_1 s_2} s_3$." By replacing the resolving signatures with their resolvent, the following grid is obtained:

	(9]			[10,14]			[15,19]			[20,25]			[26,30]			[31)		
[21)	0	□	$\overline{s_1 s_2}$	1	□	s_1	1	□	s_1	1	□	s_1	1	□	s_1	0	□	$\overline{s_1 s_2}$
[16,20]	0	□	$\overline{s_1 s_2}$	1	□	s_1	1	□	s_1	1	$\langle s_2 \rangle$	s_1	1	$\langle s_2 \rangle$	s_1	0	□	$\overline{s_1} s_2 \overline{s_3}$
[13,15]	0	□	$\overline{s_1 s_2}$	0	□	$\overline{s_1 s_2}$	0	□	$\overline{s_1 s_2}$	0	□	$\overline{s_1} s_2 \overline{s_3}$	0	□	$\overline{s_1} s_2 \overline{s_3}$	0	□	$\overline{s_1} s_2 \overline{s_3}$
[10,12]	0	□	$\overline{s_1 s_2}$	0	□	$\overline{s_1 s_2}$	0	□	$\overline{s_1 s_2}$	1	□	$\overline{s_1} s_2 s_3$	0	□	$\overline{s_1} s_2 \overline{s_3}$	0	□	$\overline{s_1} s_2 \overline{s_3}$
(9]	0	□	$\overline{s_1 s_2}$	0	□	$\overline{s_1 s_2}$	0	□	$\overline{s_1 s_2}$	0	□	$\overline{s_1 s_2}$	0	□	$\overline{s_1 s_2}$	0	□	$\overline{s_1 s_2}$

The grid array of this grid state may be reduced, since the cells on the columns corresponding to the intervals $[10, 14]$ and $[15, 19]$ are pairwise identical, i.e. they show the same G-, Q- and Z-values. These two columns can thus be collapsed into one, thus causing the merge of the corresponding intervals. The reduced state is:

	(9]			[10,19]			[20,25]			[26,30]			[31)		
[21)	0	□	$\overline{s_1 s_2}$	1	□	s_1	1	□	s_1	1	□	s_1	0	□	$\overline{s_1 s_2}$
[16,20]	0	□	$\overline{s_1 s_2}$	1	□	s_1	1	$\langle s_2 \rangle$	s_1	1	$\langle s_2 \rangle$	s_1	0	□	$\overline{s_1} s_2 \overline{s_3}$
[13,15]	0	□	$\overline{s_1 s_2}$	0	□	$\overline{s_1 s_2}$	0	□	$\overline{s_1} s_2 \overline{s_3}$	0	□	$\overline{s_1} s_2 \overline{s_3}$	0	□	$\overline{s_1} s_2 \overline{s_3}$
[10,12]	0	□	$\overline{s_1 s_2}$	0	□	$\overline{s_1 s_2}$	1	□	$\overline{s_1} s_2 s_3$	0	□	$\overline{s_1} s_2 \overline{s_3}$	0	□	$\overline{s_1} s_2 \overline{s_3}$
(9]	0	□	$\overline{s_1 s_2}$	0	□	$\overline{s_1 s_2}$	0	□	$\overline{s_1 s_2}$	0	□	$\overline{s_1 s_2}$	0	□	$\overline{s_1 s_2}$

The partitions underlying the grid states seen so far can be characterized as follows:

(i) each region of a partition is given by the Cartesian product of sets drawn from a partition of each property domain;

(ii) the database entities associated to any region of a partition are either all locked or all free in the current state;

(iii) each partition is the one with the largest regions among those satisfying the properties (i) and (ii) above.

The first two properties are evident. To see that also the third holds, it is sufficient to observe that if we make one region of a partition bigger by augmenting any set of a property domain partition, then we lose the second property, as at least one region of the resulting partition would include both locked and free entities. Partitions that satisfy condition (i) will be said to be *regular,* while those satisfying condition (ii) will be said to be *discriminating* (a certain predicate).

5.2 Discriminating partitions

In this section we will provide a lattice-theoretic formalization of the partitions presented in the previous section and of their corresponding grid states.

Definition 15: Given a database $\langle E, \mathcal{D}, \mathcal{S} \rangle$, with simple properties $\langle N_1, D_{i_1} \rangle, \ldots, \langle N_n, D_{i_n} \rangle$, the *database space* Δ is the set $D_{i_1} \times D_{i_2} \times \ldots \times D_{i_n}$. A *database partition,* or simply a *partition,* is any partition of Δ. □

In order to compare partitions with respect to their size, we introduce a relation between partitions based on a region containment criterion. Let π be the set of all partitions of a given database.

Definition 16: Given two partitions $\mathcal{H}_1$ and $\mathcal{H}_2$ in π, $\mathcal{H}_1$ is *smaller* than $\mathcal{H}_2$, $\mathcal{H}_1 \sqsubseteq \mathcal{H}_2$, if and only if for each $H_1 \in \mathcal{H}_1$ there exists $H_2 \in \mathcal{H}_2$ such that $H_1 \subseteq H_2$. $\mathcal{H}_1$ is *strictly smaller* than $\mathcal{H}_2$, $\mathcal{H}_1 \sqsubset \mathcal{H}_2$, if and only if $\mathcal{H}_1 \sqsubseteq \mathcal{H}_2$ and $\mathcal{H}_1 \neq \mathcal{H}_2$.

As it can be easily proved [13]:

Proposition 6: $(\pi, \sqsubseteq)$ is a partial order. Moreover, given any two partitions $\mathcal{H}_1 = \{A_1, A_2, \ldots, A_k\}$ and $\mathcal{H}_2 = \{B_1, B_2, \ldots, B_m\}$ in π,

$$\begin{aligned} glb(\mathcal{H}_1, \mathcal{H}_2) &= \{A_i \cap B_j \mid A_i \in \mathcal{H}_1, B_j \in \mathcal{H}_2\}, \text{ and} \\ lub(\mathcal{H}_1, \mathcal{H}_2) &= \{\cup_{A_i \in I} A_i \mid I \text{ is a smallest subset of } \mathcal{H}_1 \text{ such that,} \\ &\qquad \text{for some } J \subseteq \mathcal{H}_2,\ \cup_{A_i \in I} A_i = \cup_{B_j \in J} B_j\} \end{aligned}$$

are, respectively, the greatest lower bound and the least upper bound of $\mathcal{H}_1$ and $\mathcal{H}_2$. □

As a corollary of the last proposition, $(\pi, \sqsubseteq)$ is a lattice [14]. The greatest and smallest element of the lattice, respectively denoted by $\mathcal{H}_\top$ and $\mathcal{H}_\bot$, are readily found:

$$\begin{aligned} \mathcal{H}_\top &= \{\Delta\}, \\ \mathcal{H}_\bot &= \{\{x\} \mid x \in \Delta\}. \end{aligned}$$

One important characteristic of the partitions presented in the previous section is that they discriminate the state predicate. The next two definitions make this concept precise.

Definition 17: Given a formula ϕ of the database language $\mathcal{LS}$, the *set defined by* ϕ, $def[\phi]$, is the set of points $\langle a_1, a_2, \ldots, a_n \rangle$ of the database space such that the formula obtained by replacing in ϕ each function term $N_j(x)$ by a_j is true for all $1 \leq j \leq n$. □

Unlike the extension of a formula, the set defined by a formula does not require the interpretation of the function symbols, therefore it is independent from database states. Intuitively, a point $\langle a_1, a_2, \ldots, a_n \rangle$ is in the set defined by a formula α if, whenever an entity e takes a_j as value of the j-th property in a certain database state, then e satisfies α in that state. The relationship between the satisfiability of a formula in $\mathcal{LS}$ and the set defined by that formula is given in the following proposition.

Proposition 7: For any formula α in $\mathcal{LS}$, α is satisfiable if and only if $def[\alpha] \neq \emptyset$.
Proof: ($\leftarrow$) Let $\langle a_1, a_2, \ldots, a_n \rangle \in def[\alpha]$. Then, the database state DB such that $F_{DB}(N_j)(e) = a_j$ for some entity $e \in E_{DB}$ and for all $1 \leq j \leq n$, satisfies α. ($\rightarrow$) Conversely, if α is satisfiable, there exists a database state DB such that $\varepsilon_{DB}[\alpha] \neq \emptyset$. Let e be in $\varepsilon_{DB}[\alpha]$. Then the point $\langle F_{DB}(N_1)(e), F_{DB}(N_2)(e), \ldots, F_{DB}(N_n)(e) \rangle$ is in $def[\alpha]$, which is therefore non-empty. □

Definition 18: Given a formula ϕ of the database language and a partition $\mathcal{H}$, $\mathcal{H}$ is said to be a *discriminating partition of* ϕ if and only if for any $H \in \mathcal{H}$, either $H \subseteq def[\phi]$ or $(H \cap def[\phi]) = \emptyset$. The *discriminating function of* $\mathcal{H}$ is the total function G from $\mathcal{H}$ to $\{0, 1\}$ such that, for all $H \in \mathcal{H}$, $G(H) = 1$ if $H \subseteq def[\phi]$, and $G(H) = 0$ otherwise. □

Example 5: Let us consider our small database with two properties; the set defined by the simple atom $(N_2 \geq 2)$ is $S_1 = \{\langle n_1, n_2 \rangle \mid n_2 \geq 2\}$. A discriminating partition of this simple atom is $\{S_1, \overline{S_1}\}$, where $\overline{S_1}$ is the complement of S_1 in the database space, that is $\overline{S_1} = \{\langle n_1, n_2 \rangle | \ n_2 \leq 1\}$. The discriminating function of the partition is: $G(S_1) = 1$ and $G(\overline{S_1}) = 0$. □

The following lemma highlights useful properties of discriminating partitions.

Lemma 2: Let $\mathcal{H}_1$ and $\mathcal{H}_2$ be partitions and α and β formulae. Then:

(i) $\mathcal{H}_1$ discriminates α if and only if it discriminates $\overline{\alpha}$;

(ii) if $\mathcal{H}_1$ discriminates α and β then it discriminates $(\alpha \wedge \beta)$;

(iii) if $\mathcal{H}_2$ discriminates α and $\mathcal{H}_1 \sqsubseteq \mathcal{H}_2$, then $\mathcal{H}_1$ discriminates α.

Proof: Trivial. □

The concepts introduced so far allow us to formulate the central problem of the optimal scheduler in semantic terms: to identify, at each scheduling stage, the largest partition that discriminates the current state predicate. It is not

difficult to see that this partition, let it be $\mathcal{H}_\sigma$, is given by

$$\mathcal{H}_\sigma \quad = \quad \{def[\sigma], def[\overline{\sigma}]\},$$

where σ is the state predicate of the above tree state. To see that keeping track of $\mathcal{H}_\sigma$ may be very expensive, let us consider one of its descendants in $(\pi, \sqsubseteq)$, namely the *closed leaf nodes partition,* which for the tree state $\langle \Lambda, \Phi, q, z \rangle$ is given by:

$$\mathcal{H}_\Lambda \quad = \quad \{def[\lambda_i] \mid \lambda_i \in \Lambda\} \cup \bigcup_{\phi_j \in \Phi} def[\phi_j]$$

The analysis of the efficiency of *tree_sch* reveals (Proposition 5) that computing each λ_i has, in the worst case, an exponential cost, therefore so does the computation of $\mathcal{H}_\Lambda$ and $\mathcal{H}_\sigma$, when a DNF representation os used.

It turns out that the size of a partition is one of the factors that impacts on efficiency; another factor, indeed much more important, is the shape of the regions of the partition. The next definition introduces the special kind of partitions informally presented in the previous section, built upon partitions of the domains of the database properties. Let $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m$, $m \geq 1$, be sets of sets; we will use the following abbreviation:

$$bi\_prod(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_m) = \{A_1 \times A_2 \times \ldots \times A_m \mid A_i \in \mathcal{A}_i,\ 1 \leq i \leq m\}.$$

Definition 19: A partition of the database space is said to be *regular* if it is given by $bi\_prod(h_1, h_2, \ldots, h_n)$, where h_k is a partition of D_{i_k}, for all $1 \leq k \leq n$. □

Lemma 3: Let $\mathcal{H} = bi\_prod(h_1, h_2, \ldots, h_n)$ and $\mathcal{H}' = bi\_prod(h'_1, h'_2, \ldots, h'_n)$ be regular partitions. Then $\mathcal{H} \sqsubset \mathcal{H}'$ implies $h_i \sqsubseteq h'_i$, for all $1 \leq i \leq n$, and for at least one i $h_i \sqsubset h'_i$.
Proof: Trivial. □

Let π_r stand for the set of the regular partitions. The following proposition shows that regular partitions are closed under the operations *glb* and *lub*, hence $(\pi_r, \sqsubseteq)$ is a lattice too.

Proposition 8: If $\mathcal{H}_1 = bi\_prod(h_1, h_2, \ldots, h_n)$ and $\mathcal{H}_2 = bi\_prod(i_1, i_2, \ldots, i_n)$ are regular partitions, so are $glb(\mathcal{H}_1, \mathcal{H}_2)$ and $lub(\mathcal{H}_1, \mathcal{H}_2)$.
Proof: Extending the *glb* operator to partitions of single domains, we have that

$$glb(\mathcal{H}_1, \mathcal{H}_2) \quad = \quad bi\_prod(glb(h_1, i_1), \ldots, glb(h_n, i_n)).$$

From Proposition 6, the generic element of the $glb(\mathcal{H}_1, \mathcal{H}_2)$, is given by:

$$(h_{1j_1} \times \ldots \times h_{nj_n}) \cap (i_{1k_1} \times \ldots \times i_{nk_n}) = (h_{1j_1} \cap i_{1k_1}) \times \ldots \times (h_{nj_n} \cap i_{nk_n}),$$

where $h_{lj_l} \in h_l$ and $i_{lj_l} \in i_l$ for all $1 \leq l \leq n$. Analogously, it can be shown that

$$lub(\mathcal{H}_1, \mathcal{H}_2) \quad = \quad bi\_prod(lub(h_1, i_1), \ldots, lub(h_n, i_n)). \qquad \square$$

It should be intuitively evident, and will later be proved, that computing regular partitions is much easier than computing partitions whose regions are arbitrarily shaped. This motivates the adoption of this kind of partitions in the implementation of the optimal scheduler.

Definition 20: Given a tree state $\tau = \langle \Lambda, \Phi, q, z \rangle$ with state predicate σ, the *grids* of τ, $GR(\tau)$, are the 4-tuples $\langle \mathcal{H}, G, Q, Z \rangle$, where:

(i) $\mathcal{H}$ is a regular partition that, for each $\lambda \in \Lambda$ and $\phi \in \Phi$, discriminates λ, ϕ and $\lambda \wedge p(s)$ for each step s in $q(\lambda)$;

(ii) G is the discriminating function of σ;

(iii) Q is a total function from $\mathcal{H}$ to sequences of steps, such that for all $H \in \mathcal{H}$,

$$Q(H) \quad = \quad \begin{cases} q(\lambda)|s : (H \cap def[p(s)]) \neq \emptyset & \text{if } H \subseteq def[\lambda] \text{ for some } \lambda \in \Lambda \\ \square & \text{otherwise} \end{cases}$$

(iv) Z is a total function from $\mathcal{H}$ to signatures, such that for all $H \in \mathcal{H}$,

$$Z(H) = z(\theta), \qquad \text{where } H \subseteq def[\theta], \text{ for some } \theta \in (\Lambda \cup \Phi).$$

Each 4-tuple in $GR(\tau)$ is called a *grid state.* The grid states in $GR(\tau)$ are said to be *equivalent* to each other and to τ. $\square$

The correspondence between tree and grid states captured by the last definition is the formal counterpart of the shift in perspective discussed in the previous section. The definition requires each grid state equivalent to τ to discriminate three kinds of predicates: (1) the closed leaf node predicates, because they are the predicates output by the scheduler, so they can be considered the medium of exchange between the scheduler and the outside world; (2) the open leaf node predicates, because it is on the basis of these predicates that the predicates of the first kind are computed upon scheduling a new lock step; (3) the predicates of the form $\lambda \wedge p(s)$, because they denote the entities that will be granted at some point in the future. The second component of a grid state will permit us to distinguish regions that correspond to locked entities from regions associated to free entities. The third component is needed to keep track of queued steps; notice that since the set defined by a closed leaf node predicate may be partitioned into several regions, the queue associated to one

such region need not contain all the steps queued on the closed leaf node predicate, but only those whose predicates denote entities belonging to that region. Finally, the Z component is needed to perform the simplification following an unlock. We will apply to grid states the same notational convention adopted for tree states, taking the freedom to specify a step number for grid states and for their components, when needed.

For a given tree state, there will generally be many equivalent grid states, differing in the partition component (the other components are uniquely determined by the first one and by the definition of grids). The ordering defined on partitions can then be used to establish an ordering among equivalent grid states, in order to capture an efficiency criterion.

Definition 21: Given the grid states $\gamma_1 = \langle \mathcal{H}_1, G_1, Q_1, Z_1 \rangle$ and $\gamma_2 = \langle \mathcal{H}_2, G_2, Q_2, Z_2 \rangle$ in the grids of a given tree state τ, γ_1 is said to be *less efficient* than γ_2, $\gamma_1 \sqsubseteq_e \gamma_2$, if and only if $\mathcal{H}_1 \sqsubseteq \mathcal{H}_2$. γ_1 is *strictly less efficient* than γ_2, $\gamma_1 \sqsubset_e \gamma_2$, if and only if $\mathcal{H}_1 \sqsubset \mathcal{H}_2$. □

The efficiency relation induces a lattice structure on the grids of a given tree state, based on the structure induced on partitions by the $\sqsubseteq$ relation. To see how, for a given tree state τ, let us set:

$$\pi_r(\tau) \quad = \quad \{\mathcal{H} \in \pi_r \mid \langle \mathcal{H}, G, Q, Z \rangle \in GR(\tau)\},$$

where π_r is the set of regular partitions.

Proposition 9: For any tree state τ, $(\pi_r(\tau), \sqsubseteq)$ is a sublattice of $(\pi_r, \sqsubseteq)$.

Proof: It must be shown that for any two partitions $\mathcal{H}_1$ and $\mathcal{H}_2$ in $\pi_r(\tau)$, both $glb(\mathcal{H}_1, \mathcal{H}_2)$ and $lub(\mathcal{H}_1, \mathcal{H}_2)$ are in $\pi_r(\tau)$. As far as the former is concerned, by lemma 2(iii) $glb(\mathcal{H}_1, \mathcal{H}_2)$ discriminates the same predicates discriminated by $\mathcal{H}_1$ and $\mathcal{H}_2$, therefore it is in $\pi_r(\tau)$. On the other hand, $lub(\mathcal{H}_1, \mathcal{H}_2)$ is not in $\pi_r(\tau)$ if and only if it does not discriminate one of the predicates discriminated by $\mathcal{H}_1$ and $\mathcal{H}_2$, but, for the minimality required by the definition of lub, this may only happen if that predicate is not discriminated by one of $\mathcal{H}_1$ and $\mathcal{H}_2$, contradicting the hypothesis. □

From the last proposition and the definition of the efficiency relation, it follows that also $(GR(\tau), \sqsubseteq_e)$ is a lattice.

For the efficient implementation of our scheduler, we are interested in the maximal grid state among those equivalent to the current tree state. The following proposition gives a necessary and sufficient condition for the maximality of grid states.

Proposition 10: Let $\mathcal{H} = bi\_prod(h_1, h_2, \ldots, h_n)$ be a regular partition of a grid state $\gamma = \langle \mathcal{H}, G, Q, Z \rangle$, in $GR(\tau)$ for some tree state τ. Then γ is the

maximum of $(GR(\tau), \sqsubseteq_e)$ if and only if for no two elements h_{i_1} and h_{i_2} of a domain partition h_i, $1 \leq i \leq n$, for all regions $H_1, H_2 \in \mathcal{H}$ differing only for h_{i_1} and h_{i_2}, that is:

$$H_I \quad = \quad h_{1j_1} \times \ldots \times h_{i_I} \times \ldots \times h_{nj_n} \qquad I = 1, 2,$$

where $h_{lj_l} \in h_l$, for all $1 \leq l \leq n$, $l \neq i$, the following conditions hold:

(i) $Z(H_1) = Z(H_2)$, and

(ii) $Q(H_1) = Q(H_2)$.

Proof: ($\rightarrow$) If such elements h_{i_1} and h_{i_2} exist, then a more efficient grid than γ is easily derived. ($\leftarrow$) If γ is not maximal, then there exists a grid state $\gamma' = \langle \mathcal{H}', G', Q', Z' \rangle$ in $GR(\tau)$, with $\mathcal{H}' = bi\_prod(h'_1, \ldots, h'_n)$, such that $\gamma \sqsubset_e \gamma'$, that is $\mathcal{H} \sqsubset \mathcal{H}'$. By Lemma 3, $\mathcal{H} \sqsubset \mathcal{H}'$ implies that, for at least one i, there exist two sets h_{i_1} and h_{i_2} in h_i and a set h'_{i_1} in h'_i such that $(h_{i_1} \cup h_{i_2}) \subseteq h'_{i_1}$. Let H denote any region in $\mathcal{H}'$ that has h'_{i_1} as component, and let H_1, H_2 denote the corresponding regions in $\mathcal{H}$, having h_{i_1} and h_{i_2} as components in place of h'_{i_1}, respectively. It follows that $(H_1 \cup H_2) \subseteq H$. If H_1 and H_2 are contained in the set defined by two different leaf node predicates in Λ, the γ' would not discriminate all the predicates in $(\Lambda \cup \Phi)$, hence it would not be in $GR(\tau)$; it follows that either H_1 and H_2 are contained in $def[\lambda]$ for some $\lambda \in \Lambda$, or they are both contained in $def[\phi]$ for some $\phi \in \Phi$. So we have $Z(H_1) = Z(H_2)$. Furthermore, if it were $Q(H_1) \neq Q(H_2)$, then H_1 and H_2 would satisfy the predicate of different subsets of the steps enqueued on that predicate, hence γ' would not discriminate $(\lambda \wedge p(s))$ for a step $s \in q(\lambda)$, thus violating again condition (i) in the definition of $GR(\tau)$, which would mean that γ' is not a member of $GR(\tau)$. It follows that $Q(H_1) = Q(H_2)$. □

We will denote as $\gamma^o = \langle \mathcal{H}^o, G^o, Q^o, Z^o \rangle$ the maximum of $(GR(\tau), \sqsubseteq_e)$, also called the *optimal grid*. The last proposition suggests a method to find γ^o starting from any grid γ in $GR(\tau)$: if γ satisfies the hypotheses of the proposition, then it is the maximum; if not, the proof of the proposition indicates how to find a larger grid γ', on which the same process is iterated. Since the number of domains is finite, and so is the cardinality of each domain partition, the maximal grid can be found in a finite number of such iterations. This fact will be exploited when we need to pass from a non-optimal grid state to the optimal one.

The problem of efficiently implementing our scheduler can be posed as follows: given an input schedule sch, find γ^o_k for all $1 \leq k \leq |sch|$. We will specify the solution to this problem in an inductive way; for an arbitrary schedule sch,

Table III: Components of simple partitions

α_j	t_j
true	$\{D_{i_j}\}$
$(N_j(x) \leq d)$	$\{\{c \in D_{i_j} \mid c \leq d\}, \{c \in D_{i_j} \mid c \geq (d+1)\}\}$
$(N_j(x) \geq d)$	$\{\{c \in D_{i_j} \mid c \leq (d-1)\}, \{c \in D_{i_j} \mid c \geq d\}\}$
$(d_1 \leq N_j(x) \leq d_2)$	$\{\{c \in D_{i_j} \mid c \leq (d_1-1)\}, \{c \in D_{i_j} \mid d_1 \leq c \leq d_2\},$ $\{c \in D_{i_j} \mid c \geq (d_2+1)\}\}$

we will first identify the optimal grid state after the scheduling of the first step in *sch*, that is γ_1^o; then, we will show how the optimal state resulting from the scheduling of the first $(k+1)$ steps in *sch*, γ_{k+1}^o, can be derived from the k-th optimal state γ_k^o, both in case the $(k+1)$-th step in *sch* is a lock and an unlock step.

In order to establish the basic case, let us now consider the optimal regular partition that discriminates a simple formula.

Definition 22: Let $\beta = (\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n)$ be a formula of $\mathcal{LS}$. The *simple partition of* β, $\mathcal{H}_S^\beta$, is the regular partition $bi\_prod(t_1, \ldots, t_n)$, where each t_i is the *simple domain partition of* α_i and is given by $\{def[\alpha_i], def[\overline{\alpha_i}]\}$, for all $1 \leq i \leq n$. □

For each kind of simple atom, Table III shows the corresponding simple domain partition. Each member of these partitions is an interval, with the exception of $def[\overline{\alpha_l}]$ when α_l is an interval predicate, in which case we have the union of two intervals.

Proposition 11: For any simple formula β, the largest regular partition that discriminates β is the simple partition of β.
Proof: Suppose not. Then there is a regular partition $\mathcal{H}$ that discriminates β such that $\mathcal{H}_S^\beta \sqsubset \mathcal{H}$. By Lemma 3, there must be at least one domain partition h_k of $\mathcal{H}$ that is strictly smaller than the k-th domain partition of the simple partition of β. If β is *true*, then all α_j are *true*, hence h_k must consist of one set larger than the domain of the k-th database property, D_{i_k}. This is clearly not possible. If β has at least one simple atom α_l different from *true,* then it follows that $h_k = \{D_{i_k}\}$. In this case, it can be proved that $\mathcal{H}$ does not discriminate β, contradicting the hypotheses. □

We are now ready to give γ_1^o.

Proposition 12: For any schedule $sch = s_1 \ldots$

$$\mathcal{H}_1^o \;=\; \mathcal{H}_S^{p(s_1)}.$$

For all regions $H = h_1 \times \ldots \times h_n$ in $\mathcal{H}_S^{p(s_1)}$,

$$\begin{aligned} G_1^o(H) &= \begin{cases} 1 & \text{if } h_j = def[p(s_1)^j] \text{ for all } 1 \leq j \leq n \\ 0 & \text{otherwise} \end{cases} \\ Q_1^o(H) &= \square \\ Z_1^o(H) &= \begin{cases} `s_1 & \text{if } h_j = def[p(s_1)^j] \text{ for all } 1 \leq j \leq n \\ `\overline{s_1} & \text{otherwise} \end{cases} \end{aligned}$$

where $p(s_1)^j$ denotes the j-th atom in $p(s_1)$.

Proof: By definition of τ, the grid states in $GR(\tau_1)$ must discriminate $p(s_1)$ and $\overline{p(s_1)}$, which $\mathcal{H}_S^{p(s_1)}$ does by virtue of the previous proposition, and Lemma 2(i). Clearly G_1^o is the discriminating function of $p(s_1)$, and by definition of τ, Q and Z are properly defined. So we have that the above grid state is in $GR(\tau_1)$. The optimality of this state follows from that of $\mathcal{H}_S^{p(s_1)}$. □

The following proposition presents the derivation of the optimal grid state in the case of the scheduling of a lock step.

Proposition 13: Let $\gamma_k^o = \langle \mathcal{H}_k^o, G_k^o, Q_k^o, Z_k^o \rangle$ be the optimal grid for an input schedule sch, $|sch| > k$, whose $(k+1)$-th step s_{k+1} is a lock step. Then γ_{k+1}^o is given by:

$$\mathcal{H} = glb(\mathcal{H}_k^o, \mathcal{H}_S^{p(s_{k+1})}),$$

and for all regions $H \in \mathcal{H}$, if H' denotes the region of $\mathcal{H}_k^o$ such that $H \subseteq H'$,

$$\begin{aligned} G(H) &= \begin{cases} 1 & \text{if } H \subseteq def[p(s_{k+1})] \\ G_k^o(H') & \text{otherwise} \end{cases} \\ Q(H) &= \begin{cases} Q_k^o(H') \sqcup s_{k+1} & \text{if } H \subseteq def[p(s_{k+1})] \text{ and } G_k^o(H') = 1 \\ Q_k^o(H') & \text{otherwise} \end{cases} \\ Z(H) &= \begin{cases} cat(Z_k^o(H'), `s_{k+1}) & \text{if } G_k^o(H') = 0 \text{ and } H \subseteq def[p(s_{k+1})] \\ cat(Z_k^o(H'), `\overline{s_{k+1}}) & \text{if } G_k^o(H') = 0 \text{ and } H \cap def[p(s_{k+1})] = \emptyset \\ Z_k^o(H') & \text{if } G_k^o(H') = 1 \end{cases} \end{aligned}$$

Proof: Let us first prove that the above grid state is in $GR(\tau_{k+1})$. We must show that it satisfies the four conditions established by Definition 20. The first of these conditions requires that $\mathcal{H}$ discriminate both the predicates in $(\Lambda_{k+1} \cup \Phi_{k+1})$, and $(\lambda \wedge p(s))$ for each $\lambda \in \Lambda_{k+1}$, $s \in q(\lambda)$. From the definition of τ, we have that:

$$\begin{aligned} \Lambda_{k+1} &= \Lambda_k \cup \{\phi \wedge p(s_{k+1}) |\ \phi \in \Phi_k \text{ and } \phi \wedge p(s_{k+1}) \text{ is satisfiable}\} \\ \Phi_{k+1} &= \{\phi \wedge \overline{p(s_{k+1})} |\ \phi \in \Phi_k \text{ and } \phi \wedge \overline{p(s_{k+1})} \text{ is satisfiable}\}. \end{aligned}$$

By Lemma 2, it can be shown that $\mathcal{H}$ discriminates all the members of these two sets. For all $\lambda \in \Lambda_{k+1}$ we have three cases:

(1) if $\lambda \in \Lambda_k$ and $(\lambda \wedge p(s_{k+1}))$ is satisfiable, then $q_{k+1}(\lambda) = q_k(\lambda) \sqcup s_{k+1}$. By hypothesis and Lemma 2(ii), $\mathcal{H}$ discriminates $\lambda \wedge p(s)$ for all $s \in q_k(\lambda)$; it follows that it also discriminates $\lambda \wedge p(s_{k+1})$;

(2) if $\lambda \in \Lambda_k$ and $(\lambda \wedge p(s_{k+1}))$ is not satisfiable, then $q_{k+1}(\lambda) = q_k(\lambda)$. By hypothesis $\mathcal{H}$ discriminates $\lambda \wedge p(s)$ for all $s \in q_k(\lambda)$;

(3) if $\lambda \notin \Lambda_k$, then $q_{k+1}(\lambda) = \Box$, and no discrimination is in this case required to $\mathcal{H}$.

The second condition of Definition 20 requires that the function G above discriminate σ_{k+1}, which is given by:

$$\bigvee_{\lambda \in \Lambda_{k+1}} \lambda \equiv \bigvee_{\lambda \in \Lambda_k} \lambda \vee \bigvee_{\phi \in \Phi_k} (\phi \wedge p(s_{k+1})).$$

Now let $H \in \mathcal{H}$, $H \subseteq H' \in \mathcal{H}_k^o$. G discriminates σ_{k+1}, if and only if $G(H) = 1$ when H is included in $def[\sigma_{k+1}]$, and $G(H) = 0$ when H is not included in $def[\sigma_{k+1}]$. It follows that $G(H) = 1$ if and only if $H \subseteq def[p(s_{k+1})]$, and $G(H) = G_k^o(H')$ in all the other cases. The third condition of Definition 20 requires that, for all $H \in \mathcal{H}$:

$$Q(H) \quad = \quad \begin{cases} q(\lambda) | s : (H \cap def[p(s)]) \neq \emptyset & \text{if } H \subseteq def[\lambda] \text{ for some } \lambda \in \Lambda_{k+1} \\ \Box & \text{otherwise} \end{cases}$$

Let H' be as above. If $H \subseteq def[\lambda]$ for some $\lambda \in \Lambda_{k+1}$, then we have the same three cases analyzed above, from whose inspection the condition follows. Finally, the fourth condition of Definition 20 requires that:

$$Z(H) = z(\theta), \qquad \text{where } H \subseteq def[\theta], \text{ for some } \theta \in (\Lambda \cup \Phi).$$

If $\theta \in \Lambda_{k+1}$, then either (a) $\theta \in \Lambda_k$, in which case $Z(H)$ must be by the hypothesis $Z_k^o(H')$; or (b) θ is of the form $\phi \wedge p(s_{k+1})$, in which case $Z(H)$ must be $cat(Z_k^o(H'), `s_{k+1})$. If $\theta \in \Phi_{k+1}$, then $\theta$ is of the form $\phi \wedge \overline{p(s_{k+1})}$, in which case $Z(H)$ must be $cat(Z_k^o(H'), `\overline{s_{k+1}})$. We have thus shown that the grid state $\langle \mathcal{H}, G, Q, Z \rangle \in GR(\tau_{k+1})$. To show that it is also the maximum, it is sufficient to observe that a partition satisfies the first condition of Definition 20, if and only if it discriminates the predicates discriminated by $\mathcal{H}_k^o$ and $\mathcal{H}_S^{p(s_{k+1})}$, which, respectively by hypothesis and by Proposition 11, are the largest partitions that discriminate Λ_k, Φ_k and $p(s_{k+1})$. By definition of *glb*, it follows that $\mathcal{H}$ is the maximum of $(GR(\tau_{k+1}), \sqsubseteq_e)$. $\Box$

In the scheduling of an unlock step, the derivation of the optimal grid state is slightly more complicated, due to the possibility that a simplification of the kind illustrated in the previous section may take place. In order to import this

simplification on grids, we next define an operation on grid states analogous to the simplification of tree states.

Definition 23: Given a grid state $\gamma = \langle \mathcal{H}, G, Q, Z \rangle$ and two regions H and H' in $\mathcal{H}$, the signatures $Z(H)$ and $Z(H')$ are said to be *resolving* if and only if $Z(H) = \text{`}\xi s$ and $Z(H') = \text{`}\xi\, \overline{s}$ and $G(H) = G(H') = 0$; their *resolvent* is the signature $\text{`}\xi$. The *simplification* of γ, $\Sigma(\gamma)$, is the grid state $\langle \mathcal{H}, G, Q, Z' \rangle$, where Z' is obtained by recursively replacing resolving signatures by their resolvents. □

As expected, the simplification of equivalent grid and tree states maintains the equivalence relationship between the corresponding simplified states. This is intuitively obvious, as the simplification of a tree state just reflects the passage from a splitting tree to a smaller but equivalent tree, and the simplification of a grid state only modifies the range of the signature function. The following proposition formally captures this fact.

Proposition 14: Let $\tau = \langle \Lambda, \Phi, q, z \rangle$ be a tree state, and $\gamma = \langle \mathcal{H}, G, Q, Z \rangle$ be an equivalent grid state, that is $\gamma \in GR(\tau)$. Then $\Sigma(\gamma) \in GR(\varsigma(\tau))$.
Proof: By definition of GR, $\gamma \in GR(\tau)$ contains resolving signatures if and only if τ contains resolving open leaf node predicates. In particular, given H_1 and H_2 in $\mathcal{H}$, $Z(H_1)$ and $Z(H_2)$ are resolving if and only if ϕ_1 and ϕ_2 in Φ are resolving and $H_1 \subseteq def[\phi_1]$ and $H_2 \subseteq def[\phi_2]$. Then let $\gamma = \gamma_1, \gamma_2, \ldots, \gamma_n = \Sigma(\gamma)$ and $\tau = \tau_1, \tau_2, \ldots, \tau_n = \varsigma(\tau)$ be the sequences of grid and tree states, respectively, leading to the simplification of the original state. It can be shown, by induction on the length of the derivation, that $\gamma_i \in GR(\tau_i)$ implies $\gamma_{i+1} \in GR(\tau_{i+1})$ for all $1 \leq i \leq (n-1)$. □

We can now derive a grid state equivalent to the tree state resulting from the scheduling of an unlock step. In the following, $q \uparrow$ denotes the queue q after popping its first element.

Proposition 15: Let $\gamma_k^o = \langle \mathcal{H}_k^o, G_k^o, Q_k^o, Z_k^o \rangle$ be the optimal grid for an input schedule sch, $|sch| > k$, whose $(k+1)$-th step s_{k+1} is an unlock step, and let $s_{i_1} s_{i_2} \ldots s_{i_p}$ be the lock steps enqueued on $p(s_{k+1})$. Then the grid state γ_u given by:

$$\mathcal{H}_u \;=\; \mathcal{H}_k^o;$$

for all regions $H \in \mathcal{H}_k^o$,

$$G_u(H) \;=\; \begin{cases} 0 & \text{if } H \subseteq def[p(s_{k+1})] \text{ and } Q_k^o(H) = \square \\ G_k^o(H) & \text{otherwise} \end{cases}$$

$$Q_u(H) = \begin{cases} Q_k^o(H)\uparrow & \text{if } H \subseteq def[p(s_{k+1})] \text{ and } Q_k^o(H) \neq \Box \\ Q_k^o(H) & \text{otherwise} \end{cases}$$

$$Z_u(H) = \begin{cases} cat(Z_k^o(H), `\overline{s_{i_1}}, \ldots, `\overline{s_{i_{p-1}}}, `\overline{s_{i_p}}) & \text{if } H \subseteq def[p(s_{k+1})] \text{ and } Q_k^o(H) = \Box \\ cat(Z_k^o(H), `\overline{s_{i_1}}, \ldots, `\overline{s_{i_{l-1}}}, `s_{i_l}) & \text{if } H \subseteq def[p(s_{k+1})] \text{ and } s_{i_l} \text{ is the first element of } Q_k^o(H) \\ Z_k^o(H) & \text{otherwise} \end{cases}$$

is in $GR(\tau_u)$, where $\tau_{k+1} = \varsigma(\tau_u)$.

Proof: We must show that γ_u satisfies the four conditions of Definition 20. The proof is analogous to that of Proposition 13, and for brevity we will only show the part relative to the first condition. From the definition of τ_u we have that:

$$\begin{aligned} \Lambda_u &= \Lambda_k - \{p(s_{k+1})\} \cup \{\alpha_j | \alpha_j \text{ is satisfiable}\} \quad \text{where} \\ \alpha_j &= p(s_{k+1}) \wedge p(s_{i_j}) \wedge \bigwedge_{l=1}^{j-1} \overline{p(s_{i_l})} \\ \Phi_u &= \begin{cases} \Phi_k \cup \{\beta\} & \text{if } \beta \text{ is satisfiable} \\ \Phi_k & \text{otherwise} \end{cases} \quad \text{where} \\ \beta &= p(s_{k+1}) \wedge \bigwedge_{j=1}^{p} \overline{p(s_{i_j})}\}. \end{aligned}$$

By hypothesis, $\mathcal{H}_u$ discriminates all the predicates in Λ_k, and $(p(s_{k+1}) \wedge p(s_{i_j}))$ for all $1 \leq j \leq p$. Using a simple induction argument and Lemma 2(ii), it can be shown that $\mathcal{H}_u$ discriminates the predicates of the form:

$$\beta_j = p(s_{k+1}) \wedge \bigwedge_{l=1}^{j} \overline{p(s_{i_l})}, \qquad 0 \leq j \leq p.$$

As a consequence, and by means of an even simpler induction argument, it is shown that $\mathcal{H}_u$ discriminates the predicates:

$$\alpha_j = \beta_{j-1} \wedge p(s_{i_j}), \qquad 1 \leq j \leq p.$$

We have thus proven that $\mathcal{H}_u$ discriminates all the predicates in Λ_u and Φ_u, since β is just β_p. It remains to show that $\mathcal{H}_u$ also discriminates $(\lambda \wedge p(s))$ for all $\lambda \in \Lambda_u$ and $s \in q_u(\lambda)$. This follows immediately from the fact that $\mathcal{H}_u$ discriminates α_j and $p(s_{k+1}) \wedge p(s_{i_r})$, by means of lemma 2(ii). $\Box$

If follows from the two last propositions that the simplification of γ_u, $\Sigma(\gamma_u)$ is in $GR(\varsigma(\tau_u))$, that is in $GR(\tau_{k+1})$. However, there is no guarantee that $\Sigma(\gamma_u)$ is the maximum of $(GR(\tau_{k+1}),$
$\sqsubseteq_e)$, because γ_u and $\Sigma(\gamma_u)$ differ in their signature function, respectively Z and Z'. In passing from Z to Z', $\Sigma(\gamma_u)$ may satisfy condition (i) of Proposition 10.

This is desirable, because it is to be expected that a simplification of a splitting tree results, at least in some cases, in a simplification of the partition of the corresponding optimal grid state, and in generating γ_u and its simplified version $\Sigma(\gamma_u)$ no such simplification is involved.

As already pointed out, the optimal grid state can be obtained from $\Sigma(\gamma_u)$ by joining the elements of domain partitions that satisfy the hypotheses of Proposition 10, and repeating the process on the obtained grid state until a grid state is found that does not satisfy these hypotheses. In this way, an effective simplification of the partition takes place, which reduces the size of the current grid state.

We have now set up the theoretical background for defining an efficient implementation of the optimal scheduler, based on the notion of optimal grid state.

5.3 The scheduler

In order to maintain a strict correspondence between the theory and the implementation, the scheduler (Figure 7) has been divided into three main procedures: *grid_init,* responsible for scheduling the first step, thereby initializing the state data structure; *grid_lock,* which schedules the lock steps following the first one; and *grid_unlock,* which schedules unlock steps. These procedures are presented in the usual notation.

The state data structure employed for the implementation of the optimal scheduler consists of the following components:

- n arrays $SC_1, \ldots, SC_n$, called *scales,* one for each property of the database, in which the current partition of the property domain is maintained; the size of the array SC_j is denoted by N_j, for all $1 \leq j \leq n$; a cell of the grid array, representing a member of the current partition of the database space, is identified by an n-tuple of elements each drawn from the corresponding scale, that is $(SC_1(i_1), SC_2(i_2), \ldots, SC_n(i_n))$ where $1 \leq i_j \leq N_j$, for all $1 \leq j \leq n$;

- three functions, G, Q, and Z, giving, for each cell of the grid array, respectively, the G-value, the queue and the signature of the corresponding element of the current database partition.

Figure 8 shows the *grid_init* procedure. First, the procedure outputs the lock step s received as input. It then initializes the grid array to the simple partition of $p(s)$, as required by Proposition 12. It does this by setting each scale to the corresponding simple domain partition, as required by Definition

procedure *grid_sch* (s: **step**)
begin
if $first\_step(s)$ **then** *grid_init(s)*
else if $a(s) =$ '*lock*' **then** *grid_lock(s)*
else if $a(s) =$ '*unlock*' **then** *grid_unlock(s)*
end

Figure 7: The *grid_sch* scheduler

procedure *grid_init* (s: **step**)
begin
output(s)
for $k = 1$ **to** n **do**
  begin
  if $p(s)^k = true$ **then begin** $SC_k(1) \leftarrow D_{i_k}$; $N_k \leftarrow 1$ **end**
  else begin $SC_k(1) \leftarrow def[p(s)^k]$; $SC_k(2) \leftarrow def[\overline{p(s)^k}]$; $N_k \leftarrow 2$ **end**
  end
for $j_1 = 1$ **to** N_1 **do**
...
for $j_n = 1$ **to** N_n **do**
  begin
  if $SC_1(j_1) = def[p(s)^1]$ **and** ... **and** $SC_n(j_n) = def[p(s)^n]$ **then begin**
    $G(SC_1(j_1), \ldots, SC_n(j_n)) \leftarrow 1$; $Z(SC_1(j_1), \ldots, SC_n(j_n)) \leftarrow$'s
    end
  else begin
    $G(SC_1(j_1), \ldots, SC_n(j_n)) \leftarrow 0$; $Z(SC_1(j_1), \ldots, SC_n(j_n)) \leftarrow$ '$\overline{s}$
    end
  $Q(SC_1(j_1), \ldots, SC_n(j_n)) \leftarrow \Box$
  end
end

Figure 8: The scheduling of the first step

procedure *grid_lock* (s: **step**)
begin
for $k = 1$ **to** n **do** $merge(SC_k, N_k, def[p(s)^k], def[\overline{p(s)^k}], SC'_k, N'_k)$
for $j_1 = 1$ **to** N'_1 **do**
...
for $j_n = 1$ **to** N'_n **do**
  begin
  $k_1 \leftarrow includes(SC_1, SC'_1(j_1))$
  ...
  $k_n \leftarrow includes(SC_n, SC'_n(j_n))$
  if $contained(p(s)^1, SC'_1(j_1))$ **and** ... **and** $contained(p(s)^n, SC'_n(j_n))$ **then begin**
    $G'(SC'_1(j_1), \ldots, SC'_n(j_n)) \leftarrow 1$
    if $G(SC_1(k_1), \ldots, SC_n(k_n)) = 1$ **then begin**
      $Q'(SC'_1(j_1), \ldots, SC'_n(j_n)) \leftarrow Q(SC_1(k_1), \ldots, SC_n(k_n)) \sqcup s$
      $Z'(SC'_1(j_1), \ldots, SC'_n(j_n)) \leftarrow Z(SC_1(k_1), \ldots, SC_n(k_n))$
      end
    else begin
      output$(step(SC'_1(j_1), \ldots, SC'_n(j_n)))$
      $Q'(SC'_1(j_1), \ldots, SC'_n(j_n)) \leftarrow \Box$
      $Z'(SC'_1(j_1), \ldots, SC'_n(j_n)) \leftarrow cat(Z(SC_1(k_1), \ldots, SC_n(k_n)), `s)$
      end
    end
  else begin
    $G'(SC'_1(j_1), \ldots, SC'_n(j_n)) \leftarrow G(SC_1(k_1), \ldots, SC_n(k_n))$
    $Q'(SC'_1(j_1), \ldots, SC'_n(j_n)) \leftarrow Q(SC_1(k_1), \ldots, SC_n(k_n))$
    if $G(SC_1(k_1), \ldots, SC_n(k_n)) = 1$ **then** $Z'(SC'_1(j_1), \ldots, SC'_n(j_n)) \leftarrow Z(SC_1(k_1), \ldots, SC_n(k_n))$
    else $Z'(SC'_1(j_1), \ldots, SC'_n(j_n)) \leftarrow cat(Z(SC_1(k_1), \ldots, SC_n(k_n)), `\overline{s})$
    end
  end
end

Figure 9: The scheduling of lock steps

22. Finally, *grid_init* assigns to G, Q and Z the proper value, as established by Proposition 12.

The *grid_lock* procedure, presented in Figure 9, computes the new optimal grid in a fresh state data structure, whose components are denoted by priming the variables of the current one. By Proposition 13, the partition of the new optimal grid is the greatest lower bound of the current partition and the simple partition of $p(s)$, and is given, according to Proposition 8, by the *bi_prod* of the greatest lower bound of the corresponding domain partitions, to be computed as established by Proposition 6. This computation is the task of the *merge* procedure (Figure 10), which takes as input a scale SC_k with the current k-th domain partition, its size, and the elements of the k-th simple domain partition, which have been denoted as $interval_1$ and $interval_2$, even though one of them may in fact be the union of two intervals (see Table III). The two other parameters of *merge* are the returned new scale and its size. *merge* checks each member of SC_k against both $interval_1$ and $interval_2$ to ascertain whether the intersection yields the empty set. If not, the intersection set is assigned to the new scale. Since also the members of scales are unions of disjoint intervals, the task of *merge* is almost trivial.

Having computed the components of the new partition, *grid_lock* examines each element of this partition to update its G- Q- and Z- values as required by Proposition 13. For each such element H, *grid_lock* needs to know:

(1) What is the region of the old partition containing H, that is H' in Proposition 13. This region is given by $(SC_1(k_1), \ldots, SC_n(k_n))$, where each k_i is computed by the *includes* procedure (not presented), which finds the member of the old scale SC_i containing the i-th component of H.

(2) Whether H is in the region defined by the predicate of the step being scheduled. Again, this is done scale by scale applying the *contained* predicate, which checks whether the interval defined by a given atom contains an element of a domain partition. The procedure implementing the *contained* predicate is not presented.

In updating the state data structure, *grid_lock* strictly follows the instructions implicitly given in Proposition 13, with one addition: whenever an element of the new partition is found to be contained in the set defined by the input predicate and to come from a 0 G-valued region of the old partition, then, as expected, a lock step is output. The predicate of this lock step is obtained as the conjunction of the atoms corresponding to the domain partition elements of the element in question. This operation is accomplished by the *step* procedure, not presented.

```
procedure merge (SC, N, interval_1, interval_2, SC', N')
split: boolean;   increment: integer
begin
increment ← 0
for j = 1 to N do
  begin
  split ← false
  if (SC(j) ∩ interval_1) ≠ ∅ then begin
      split ← true;   SC'(j + increment) ← SC(j) ∩ interval_1
      end
  if (SC(j) ∩ interval_2) ≠ ∅ then begin
      if split then increment ← increment + 1
      SC'(j + increment) ← SC(j) ∩ interval_2
      end
  end
N' ← N + increment
end
```

Figure 10: The procedure computing the greatest lower bound of two domain partitions

Finally, the *grid_unlock* procedure is given in Figure 11. This procedure first computes the grid state γ_u defined in Proposition 15, then its simplification $\Sigma(\gamma_u)$, by means of the *grid_simplify* procedure, and finally the reduced state, which the *grid_reduce* procedure obtains by applying the result of Proposition 10, as already explained. Neither *grid_simplify* nor *grid_reduce* are presented, as they are not conceptually relevant.

In order to compute γ_u, *grid_unlock* first collects in the *pending* set the steps which are enqueued in the regions defined by the predicate of the unlocked step. The names of these steps are needed to compute the signatures of the regions involved in the rescheduling process, and are denoted as $s_{i_1} s_{i_2} \ldots s_{i_p}$ in Proposition 15. The members of *pending* are then sorted according to their arrival order by the *sort* function. *grid_unlock* identifies the regions of the current partition involved in the unlock in the same way as *grid_lock*, that is by scanning the whole grid array and testing the single components of each region against the corresponding atom of the input predicate. For each such region, *grid_unlock* updates the G, Q and Z functions as required by Proposition 15, with one addition: whenever an element is found to be contained in the

set defined by the input predicate and to have a non-empty queue, then, as expected, a lock step is output, so rescheduling the first of the steps pending on the region. This step is then dequeued by *pop*-ing the queue of the region. To compute signatures, two functions are used:

- *select*, returning the subsequence given as first argument up to and excluding the element given as second argument. When the second argument is *all*, all of the first argument is returned;
- *neg*, taking as input a sequence of steps and returning the string obtained by concatenating the names of the steps, negated.

The correctness of *grid_sch* follows from that of *tree_sch,* as these schedulers grant the same database entities, although possibly grouped in a different way.

Let us now consider the efficiency of *grid_sch.* We recall that the efficiency of a scheduler is measured with respect to the size of the state data structure, and the number of steps needed to schedule an input step.

The state data structure used by *grid_sch* is given by the scales $SC_1, SC_2, \ldots, SC_n$ and the functions G, Q and Z. The values taken by these functions are clearly polynomially sized with respect to the input steps, but their domain is the grid array representing the current partition, whose size depends on the size of the scales. We must thus prove that both the size of each scale, that is N^i for all $1 \leq i \leq n$, and the size of each element of the scales are polynomially bounded. In order to obtain these two results simultaneously, we will consider the number of intervals in which each property domain is partitioned, letting it be NI^j for the j-th domain partition. Since $N^j \leq NI^j$, any limitation to the latter will also apply to the former.

Proposition 16: For all $1 \leq j \leq n$, NI^j is at any moment bounded by a polynomial in the number of steps that have arrived so far.
Proof: Table III shows that $NI^j_1 \leq 3$. If the $(m+1)$-th step s of the schedule being processed is a lock step, each interval in SC_j obtained at step m is intersected with the intervals in

$$\{def[p(s)^j], def[\overline{p(s)^j}]\}.$$

In the worst case, $p(s)^j$ is an interval predicate whose boundaries are different from those of the intervals in SC_j, so that $NI^j_{m+1} \leq NI^j_m + 2$. If s is an unlock step, the size of the grid can only be decreased by a simplification, therefore $NI^j_{m+1} \leq NI^j_m$. By a simple induction argument, it can be shown that from

$$NI^j_1 \leq 3$$
$$NI^j_{m+1} \leq NI^j_m + 2$$

procedure *grid_unlock* (s: **step**)
s': **step**; *pending*: **set of step**
begin
$pending \leftarrow \emptyset$
for $j_1 = 1$ **to** N_1 **do**
...
for $j_n = 1$ **to** N_n **do**
if $contained(p(s)^1, SC_1(j_1))$ **and** ... **and** $contained(p(s)^n, SC_n(j_n))$ **then**
    $pending \leftarrow pending \cup Q(SC_1(j_1), \ldots, SC_n(j_n))$
$pending \leftarrow sort(pending)$
for $j_1 = 1$ **to** N_1 **do**
...
for $j_n = 1$ **to** N_n **do**
  begin
  if $contained(p(s)^1, SC_1(j_1))$ **and** ... **and** $contained(p(s)^n, SC_n(j_n))$ **then**
    if $Q(SC_1(j_1), \ldots, SC_n(j_n)) = \Box$ **then begin**
    $G(SC_1(j_1), \ldots, SC_n(j_n)) \leftarrow 0$
    $Z(SC_1(j_1), \ldots, SC_n(j_n)) \leftarrow cat(Z(SC_1(j_1), \ldots, SC_n(j_n)), neg(select(pending, all)))$
    end
    else begin
    output$(step(SC_1(j_1), \ldots, SC_n(j_n)))$
    $s' \leftarrow pop(Q(SC_1(j_1), \ldots, SC_n(j_n)))$
    $Z(SC_1(j_1), \ldots, SC_n(j_n)) \leftarrow cat(Z(SC_1(j_1), \ldots, SC_n(j_n)), neg(select(pending, s')), `s')$
    end
  end
grid_simplify; *grid_reduce*
end

Figure 11: The scheduling of unlock steps

it follows that $NI_m^j \leq (2 \cdot m + 1) = O(m)$, for all j, $1 \leq j \leq n$. □

As a corollary of the last proposition, we have that the size of the grid array, N, satisfies the following condition:

$$N = \prod_{j=1}^{n} N^j \leq \prod_{j=1}^{n} (2 \cdot m + 1) = O(m^n),$$

and is therefore polynomial in the size of the input, m.

As far as the second measure of efficiency is concerned, we can state the following.

Proposition 17: The number of steps needed to schedule an input step is at any moment bounded by polynomial in the number of steps that have been scheduled so far.

Proof: Both in case of a lock and an unlock step, the scheduler examines the whole grid array, whose size has been shown to be polynomial in the number of steps that have been scheduled so far. For each element examined, *grid_sch* performs operations whose complexity is linear in the product of the size of the largest queue and the size of the current scales. On the one hand, *grid_lock* performs n calls to the *includes* function, which is a linear scanning of a scale, and n checks of the *contained* predicate, which requires a containment check between intervals. On the other hand, *grid_unlock* examines the grid array twice, the first time to collect pending steps, the second time to update the G, Q and Z functions, for which n calls to the *contained* predicate are required each time. The efficiency of *grid_sch* thus follows. □

A number of speed-up devices can be used to efficiently implement the *grid_lock* and *grid_unlock* procedures, which have been shaped in a way that makes the efficiency analysis of *grid_sch* easier.

The somewhat disturbing fact remains that the size of the grid array may be exponential in the number of properties of the database, n. The consequent disappointment can be mitigated by the fact that our analysis: (a) does not take into account the simplifications allowed by the scheduling of unlock steps, and (b) assumes that in the scheduling of a lock step every linear scale increases by two elements. The former factor is especially expected to play a crucial role in maintaining the actual size of the grid array at a reasonable level. Furthermore, the number of database properties is constant in time and is not expected to be significantly high, say in the order of units rather than dozens. While the former fact allows the designer of the scheduler to be aware of the problem right from the start and so devise the necessary measures, it is the latter fact that gives the decisive plausibility to the whole method.

5.4 From $\mathcal{LS}$ to $\mathcal{L}$

We now consider how useful extensions to the expressive power of $\mathcal{LS}$ impact on the efficiency of *grid_sch*.

5.4.1 DNF

The first extension concerns the inclusion in the language of DNF predicates. In this case, for any input step s, we will have:

$$p(s) = \phi = \bigvee_{j=1}^{m} \phi_j, \qquad \text{for } m \geq 1,$$

where each ϕ_j is a formula of $\mathcal{LS}$. Let us call $\mathcal{LS}^+$ this extended version of $\mathcal{LS}$. It is not difficult to see that the scheduling of s is equivalent to the scheduling of the sequence of steps $s_1 s_2 \ldots s_m$, where:

$$p(s_j) = \phi_j, \qquad \text{for all } 1 \leq j \leq m.$$

Therefore, the complexity of scheduling s is given by the sum of the complexity of scheduling each s_j, hence it is of the same order. From the efficiency of *grid_sch* on $\mathcal{LS}$ that of *grid_sch* on $\mathcal{LS}^+$ will follow.

5.4.2 Aggregation

A useful feature of $\mathcal{L}$ that is lost in $\mathcal{LS}$ is aggregation, expressed through function terms of the form $f_1(f_2(\ldots f_m(x)\ldots))$, where each f_j is a function symbol interpreted, with the exception of f_1, as a database complex property. By allowing aggregation in input predicates, our scheduler would be able to handle predicates like:

$$Lives\_in(x) = \text{Pisa} \ \wedge \ Age(Best\_friend(x)) \leq 40,$$

denoting the entities who live in Pisa and whose best friend is at most 40 years old. Predicates of this kind address more than one entity: for instance, the above predicate refers to two entities, one denoted by the variable x, the other denoted by the atomic function term $Best\_friend(x)$. In fact, the above formula is equivalent to:

$$Lives\_in(x) = \text{Pisa} \ \wedge (\exists y)(y = Best\_friend(x) \ \wedge \ Age(y) \leq 40).$$

In order to represent the set defined by this formula, two database spaces are needed: one corresponding to the property values of x, the other to the property values of y. This fact can be expressed in a general and formal way by extending Proposition 7, and showing that a formula containing aggregations is satisfiable

if and only if the set defined in the extended space is not empty. Even if the nesting of function terms were limited to a maximum k, with m complex properties in the database there would be potentially k^m database spaces to be kept track of, i.e. k^m grid arrays for each grid state. Needless to say, this is not affordable in any reasonable database system.

5.4.3 Complex atoms

Another feature of $\mathcal{L}$ that $\mathcal{LS}$ lacks is the possibility of expressing complex atoms, that is formulae of the kind:

$$Spends(x) \leq Earns(x),$$

denoting the individuals who spend no more than they earn. Resorting to the geometrical interpretation of predicates introduced in Section 5.1, we can see that complex atoms in general define regions shaped as possibly infinite polygons. The representation of these shapes as sums of rectangles, although geometrically possible, may require as many rectangles as the points in a property domain (this is the case of the above formula). It follows that the introduction of complex atoms leads grid states to an unmanageable size.

6 Conclusions

This paper gives three main results. First, it defines the optimal splitting strategy, that is the scheduling policy that allows the maximum level of concurrency when dealing with predicate locking. This strategy is formalized by means of tree states, and proved correct. Second, it defines a language that allows an effective implementation of the optimal splitting strategy. In order to show this, the theory of discriminating partition is developed and used as a basis for the implementation of an optimal predicate locking scheduler. Third, it shows that any significant extension to the expressive power of the predicate language prevents the application of the grid method for implementing the optimal splitting strategy. This sheds light on one side of the trade-off between efficiency and concurrency involved in the design of predicate locking schedulers, namely the side where concurrency is given the highest priority.

It is still an open problem whether there exists an efficient implementation of the optimal splitting strategy for a predicate language more powerful than $\mathcal{LS}^+$.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[2] S. Böttcher, M. Jarke, and J. W. Schmidt. Adaptive predicate managers in database systems. In *Proceedings of 12th International Conference on Very Large Data Bases*, pages 21–29, Kyoto, Japan, 1986.

[3] C.-L. Chang and R. C. T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York, NY, 1973.

[4] C. Elkan. A decision procedure for conjunctive query disjointness. In *Proceedings of the Eighth Annual ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 134–139, Philadelphia, PA, 1989.

[5] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York, NY, 1972.

[6] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *ACM Communications*, 19:624–633, 1976.

[7] H. B. Hunt and D. J. Rosenkrantz. The complexity of testing predicate locks. In *Proceedings of ACM-SIGMOD 1979 International Conference on Management of Data*, pages 127–133, Boston, MA, 1979.

[8] A. Klug. Locking expressions for increased database concurrency. *Journal of the ACM*, 30(1):36–54, 1983.

[9] A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35(1):146–150, 1988.

[10] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.

[11] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, Rockville, MA, 1986.

[12] D. J. Rosenkrantz and H. B. Hunt. Processing conjunctive predicates and queries. In *Proceedings of 6th International Conference on Very Large Data Bases*, pages 64–74, Montreal, Quebec, 1980.

[13] K.A. Ross and C.R.B. Wright. *Discrete Mathematics*. Prentice Hall, Englewood Cliffs, NJ, 1985.

[14] D. E. Rutherford. *Introduction to Lattice Theory*. Oliver & Boyd Ltd., Edinburgh, UK, 1965.

[15] J. M. Smith and D. C. P. Smith. Database abstractions: Aggregation. *Communications of the ACM*, 20(6):403–413, 1977.