ABSTRACT
This paper is a review of recent theoretical work on the problems which arise when many users access the same database. For a detailed exposition of this material, the interested reader is referred to a forthcoming monograph [Pa4].

1. Introduction
Database concurrency control studies the problems that arise when many programs access and update the same data simultaneously. There is now a vast applied literature on database concurrency control (see [BG] for a review and references). More recently, a theory of concurrency control has started to emerge. The goals of this theoretical work have been compatible with the goals of the theory of computation in general:
(a) To study the whole spectrum of possible approaches to the computational problem in hand, and (b) To show rigorously the limitations of these approaches.

The concept of programs with shared data is certainly not new in computer science. Concurrent programs and communicating processes are already well-studied. That theory, however, concerns itself with a set of programs that are meant to run together, and shared data are in part means of communication and synchronization. Correctness is a collective property of the programs. In contrast, in database concurrency control we have programs that were meant to run in isolation, and were designed to be individually correct. As a consequence, an interleaved execution of such programs may fail to be correct. The canonical example is the pair of two-step programs updating the shared integer variables $x$ and $y$.

\begin{verbatim}
begin
  x := x + 1;
  y := y - 1
end
\end{verbatim}

and
begin x:=2*x;
y:=2*lT
end.

Each of these programs preserves the invariant "x+y=0". However, certain interleaved executions of their steps fail to do so. (For example, try the execution in which the two steps of the first program occur between the two steps of the second). The problem in concurrency control is to avoid such "incorrect" executions, while allowing the correct ones. This is not achieved by rewriting the programs to include synchronization, but by designing algorithms which monitor the execution and intervene to change the order of execution whenever necessary, so that the resulting concurrent execution is correct.

The algorithm which achieves this is called the scheduler. The scheduler operates on-line on a shuffled execution sequence from the individual programs. Such sequences are called schedules. The output of the scheduler is another schedule, which is guaranteed to preserve the invariants. (See Figure 1)

On the surface the problem may be reminiscent of program verification. After all, the scheduler must verify that a certain interleaved execution of some programs preserves an invariant. There are very strong reasons, however, which make the program verification approach inappropriate:

1. First, the scheduler must arrive at decisions at a speed comparable to the execution of the programs. This rules out realistically any automatic verification.

2. The precise semantics of the programs are not available to the scheduler. What is submitted to the scheduler is a sequence of accesses to the shared variables. Local variable computations are hidden from the scheduler. Even the invariants are usually not known.

These arguments point towards the more syntactic approach, which has been the framework of concurrency control theory. We introduce the basic concepts in the next section. Section 3 gives a closer look, to schedulers and their properties. Section 4 examines a class